Emanuela Cerchez, Marinel Şerban

PROGRAMAREA ÎN LIMBAJUL C/C++ PENTRU LICEU

Ediția a II-a

POLIROM

Emanuela Cerchez (n. 12.04.1968, la Iaşi) este absolventă a Facultăţii de Matematică, secţia Informatică (1990), şi a Seminarului pedagogic postuniversitar (1997), profesoară de informatică (grad didactic I), membră în Comisia Naţională de Informatică. Autoarea a mai publicat la Editura Polirom: *Internet. Manual pentru liceu* (2000, avizat MEN), *Informatica. Manual pentru clasa a X-a* (coautor Marinel-Paul Şerban, 2000, avizat MEN), *PC. Pas cu pas* (coautor Marinel-Paul Şerban, 2001), *Informatica pentru gimnaziu* (coautor Marinel-Paul Şerban, 2002, avizat MEC), *Informatica. Culegere de probleme pentru liceu* (2002).

Marinel-Paul Şerban (n. 29.06.1950, la Arad) este absolvent al Facultăţii de Matematică-Mecanică, Universitatea din Timişoara (1973), specializare postuniversitară în informatică (1974), profesor de informatică (grad didactic I), membru în Comisia Naţională de Informatică. De acelaşi autor, la Editura Polirom au apărut: *Informatica. Manual pentru clasa a X-a* (coautoare Emanuela Cerchez, 2000, avizat MEN), *PC. Pas cu pas* (coautoare Emanuela Cerchez, 2001), *Informatica pentru gimnaziu* (coautoare Emanuela Cerchez, 2002, avizat MEC).
Autorii au o bogată experienţă în pregătirea de performanţă a elevilor: susţin cursuri la Centrul de Pregătire a Tinerilor Capabili de Performanţă din Iaşi, desfăşoară probleme pentru olimpiadele şi concursurile naţionale şi judeţene de informatică, susţinând activitatea de pregătire a lotului naţional de informatică, precum şi programul de pregătire de performanţă în informatică *.campion.*

© 2005 by Editura Polirom

www.polirom.ro

Editura Polirom
Iaşi, B-dul Carol I nr. 4, P.O. BOX 266, 700506
Bucureşti, B-dul I.C. Brătianu nr. 6, et. 7, ap. 33; O.P. 37, P.O. BOX 1-728, 030174

Descrierea CIP a Bibliotecii Naţionale a României:

CERCHEZ, EMANUELA
Programarea în limbajul C/C++ / Emanuela Cerchez, Marinel Şerban. - Iaşi: Polirom, 2005
ISBN: 937-681-868-3

I. Şerban, Marinel

004.43 C
004.43 C++

Printed in ROMANIA

Emanuela Cerchez, Marinel Şerban

Programarea în limbajul C/C++ pentru liceu

POLIROM
2005

Cuprins

1. Elemente de bază ale limbajului C/C++ 11

1.1. ***Noţiuni introductive*** 11

Evoluţia limbajelor de programare 11

1.2. ***Setul de caractere*** 12

1.3. ***Identificatori*** 12

1.4. ***Cuvinte rezervate*** 13

1.5. ***Comentarii*** 13

1.6. ***Separatori*** 14

1.7. ***Structura generală a unui program C/C++*** 15

1.8. ***Tipuri de date standard*** 16

Tipul de date `int` 16

Tipul `char` 17

Tipuri reale 19

Tipul `void` 19

1.9. ***Variabile*** 20

1.10. ***Preprocesare*** 21

1.11. ***Utilizarea funcţiilor din bibliotecile standard*** 22

1.12. ***Citirea/scrierea datelor*** 24

1.13. ***Citiri şi scrieri în limbajul C++*** 24

1.14. ***Citiri şi scrieri în limbajul C*** 26

Citirea datelor cu format specificat 26

Citirea caracterelor 29

Afişarea datelor cu format 29

1.15. ***Expresii*** 31

Operatori aritmetici 32

Operatori de incrementare/decrementare 32

Operatori relaţionali 33

Operatori de egalitate 33

Operatori logici globali 34

Operatori logici pe biţi 34

Operatori de atribuire 36

Operatori condiţionali 37

Operatorul de referenţiere (adresă) 37
Operatorul de conversie explicită 37
Operatorul de determinare a dimensiunii 38
Operatorul virgulă 38
Evaluarea expresiilor 39
Tabelul priorităţii operatorilor 40
1.16. *Aplicaţii* 40
Disc 40
Compus chimic 41
Triunghi 42
Schimb 43
Leul şi iepuraşul 44
Bazin 45
1.17. *Probleme propuse* 45
2. Instrucţiunile limbajului C/C++ 50
2.1. *Instrucţiunea expresie* 50
2.2. *Instrucţiunea compusă* 50
2.3. *Instrucţiunea* `if` 51
2.4. *Instrucţiunea* `switch` 51
2.5. *Instrucţiunea* `break` 53
2.6. *Instrucţiunea* `while` 53
2.7. *Instrucţiunea* `do-while` 54
2.8. *Instrucţiunea* `for` 55
2.9. *Aplicaţii* 56
Modul 56
Paritate 57
Ecuaţia de gradul al II-lea 57
Sumă 58
Aria triunghiului 58
Secvenţă 59
Calcul 62
Numărare 64
Putere 66
Numere 67
Coduri şi caractere 68
Secvenţă simetrică 69
Maxim 70
Media aritmetică 72
Suma cifrelor 73
Perechi 73

Divizori 74
Număr prim 75
Descompunere în factori primi 76
Cel mai mare divizor comun 77
Gard 78
Exponent 79
Termen Fibonacci 80
Existență 81
Număr divizori primi 82
Vocale 82
Expresie 83
Radical 84
Relație 84
Sumă de două numere prime 85
Progres 86
Tabla adunării 87
Semne 88
Excursie 88
2.10. *Probleme propuse* 90
3. Fișiere 102
3.1. *Noțiuni introductive* 102
3.2. *Fișiere text în limbajul C++* 103
Declararea fișierelor 103
Deschiderea fișierelor 103
Citirea datelor dintr-un fișier 105
Scrierea datelor într-un fișier 105
Operații de test 106
Închiderea unui fișier 107
3.3. *Fișiere text în limbajul C* 107
Declararea fișierelor 107
Deschiderea fișierelor 108
Citirea datelor dintr-un fișier 109
Scrierea datelor într-un fișier 109
Închiderea fișierelor 109
3.4. *Probleme propuse* 111
4. Tipuri structurate de date 112
4.1. *Tablouri* 112
4.2. *Prelucrări elementare pe vectori* 114
Citirea unui vector 114
Afișarea unui vector 114

Determinarea elementului maxim/minim dintr-un vector ... 114
Media aritmetică a elementelor strict pozitive ... 115
Inversarea ordinii elementelor din vector ... 115
Verificarea unei proprietăți ... 116
Căutarea unui element într-un vector ... 117
Sortare ... 118
Interclasare ... 120
4.3. *Prelucrări elementare cu matrice* ... 121
Citirea unei matrice ... 121
Afișarea unei matrice ... 121
Parcurgerea unei matrice pe linii ... 122
Parcurgerea unei matrice pe coloane ... 122
4.4. *Prelucrări elementare specifice matricelor pătratice* ... 122
Parcurgerea diagonalelor ... 122
Parcurgerea elementelor de sub diagonala principală ... 123
4.5. *Șiruri de caractere* ... 123
4.6. *Citirea și afișarea șirurilor de caractere* ... 124
Citirea unui șir de caractere în limbajul C++ ... 124
Citirea unui șir de caractere în limbajul C ... 125
Afișarea unui șir de caractere în limbajul C++ ... 126
Afișarea unui șir de caractere în limbajul C ... 126
4.7. *Tipul pointer* ... 127
Declararea unui pointer de date ... 127
Operații cu pointeri ... 128
4.8. *Legătura dintre pointeri și tablouri* ... 130
Utilizarea funcțiilor standard de lucru cu șiruri de caractere ... 131
4.9. *Tipul de date* `struct` ... 137
Declararea unui tip `struct` ... 137
Referirea la un câmp al unei structuri ... 138
Inițializarea unei structuri ... 139
Citirea și afișarea structurilor ... 139
Operații cu structuri ... 139
4.10. *Asocierea unui nume pentru un tip de date* ... 139
4.11. *Aplicații* ... 140
Temperaturi ... 140
Inserare medii ... 141
Combinare vectori ... 141
Ciurul lui Eratostene ... 143
Copii ... 144
Perechi ... 145
Repetiție ... 146

Eliminare 148
Permutare ciclică 149
Generarea mulțimii 150
Schema lui Horner 152
Subsecvență de sumă maximă 153
Cărți 154
Marcare 155
Depozit 156
Problema celebrității 157
Cadrane 158
Matrice 159
Situație școlară 161
Pseudodiagonale 164
Submulțimi cu sume egale 167
Dreptunghi de sumă maximă 168
Secvențe 170
Problema spectacolelor 172
Problema rucsacului 173
Generare de submulțimi 175
Generare elemente produs cartezian 176
Expresie 177
Reactivi 178
Furnici 181
Apariții cifră 183
Examen de capacitate 184
Aparițiile unui cuvânt 185
Propoziție 186
4.12. *Probleme propuse* 187
5. Stiva și coada 201
5.1. *Stiva* 201
Care este utilitatea stivelor? 202
Cum implementăm o stivă? 202
5.2. *Coada* 204
Care este utilitatea unei cozi? 205
Cum implementăm o coadă? 205
5.3. *Aplicații* 207
Depou 207
Paranteze 208
Manna-Pnueli 210
Caroiaj 211
5.4. *Probleme propuse* 213

6. Funcţii 218
6.1. *Subprograme în limbajul C/C++* 219
6.2. *Definiţia unei funcţii* 219
6.3. *Declararea funcţiilor* 221
Utilitatea declaraţiilor 222
Biblioteci de funcţii şi fişiere antet 222
6.4. *Apelul funcţiilor* 223
6.5. *Transferul parametrilor prin referinţă* 226
Utilizarea pointerilor ca parametri 226
Tipul referinţă 227
6.6. *Variabile globale şi variabile locale* 231
Variabilele globale 232
Variabilele locale 232
Regula de omonimie 232
Operatorul de rezoluţie 233
6.7. *Specificatori de clasă de memorare* 233
6.8. *Parametrii funcţiei* `main()` 234
6.9. *Caracteristici specifice funcţiilor C++* 236
Funcţii inline 236
Funcţii cu parametri impliciţi 236
Supraîncărcarea funcţiilor 237
Funcţii şablon 238
6.10. *Proiecte* 240
6.11. *Aplicaţii* 242
Expresie 242
Find & Replace 243
Cel mai mare divizor comun cu descompunere în factori primi 245
Incluziune 247
Operaţii cu numere naturale mari 248
Secvenţă regulată 252
6.12. *Probleme propuse* 255
Anexe 264
1. *Tabela codurilor ASCII* 264
2. *Cuvintele cheie C/C++* 265
3. *Sisteme de numeraţie* 266
Operaţii de conversie 266
4. *Organizarea logică a memoriei* 268
5. *Reprezentarea datelor în memorie* 269
6. *Etapele dezvoltării programelor* 270

Soluții și indicații 272
1. *Elemente de bază ale limbajului C/C++* 272
2. *Instrucțiunile limbajului C/C++* 272
3. *Fișiere* 278
4. *Tipuri structurate de date* 278
5. *Stiva și coada* 285
6. *Funcții* 286
Bibliografie 293

1. Elemente de bază ale limbajului C/C++

1.1. *Noțiuni introductive*

Un limbaj de programare reprezintă un mijloc de comunicare între programator şi calculator. Pentru a defini cu rigurozitate un limbaj de programare, trebuie să descriem trei aspecte ale limbajului:

- *sintaxa* – reprezintă totalitatea regulilor pe baza cărora se obțin elementele limbajului;
- *semantica* – defineşte semnificația construcțiilor sintactic corecte (a elementelor limbajului);
- *pragmatica* – defineşte modul de utilizare a elementelor limbajului.

Observați că aceste trei aspecte se studiază şi în cazul unui limbaj natural (de exemplu, limba română). Dar spre deosebire de limba română, unde, dacă facem o greşeală gramaticală sau utilizăm greşit o expresie într-un context, sunt şanse destul de mari ca interlocutorul să ne înțeleagă, atunci când comunicăm cu un calculator, trebuie să respectăm cu exactitate regulile de comunicare. Calculatorul nu face presupuneri, nu ghiceşte ceea ce ați fi dorit să-i spuneți. Dacă nu vă „exprimați" corect, nu veți obține de la calculator decât mesaje de eroare.

Descrierea unui algoritm într-un limbaj de programare se numeşte *program*.

Evoluția limbajelor de programare

Calculatorul, la nivel intim, nu „cunoaşte" decât un singur mod de comunicare – limbajul procesorului cu care este dotat, denumit *cod-maşină*. Programarea în acest limbaj este dificilă şi necesită cunoştințe detaliate despre procesorul respectiv. Ca urmare, s-au dezvoltat mai întâi limbajele de asamblare (care necesitau în continuare cunoştințe legate de procesorul calculatorului, dar erau mai uşor de utilizat) şi apoi limbaje de programare de nivel înalt (care nu necesită cunoştințe detaliate referitoare la structura calculatorului pe care se va executa programul şi utilizează notații asemănătoare limbajului matematic sau limbajului natural).

De-a lungul timpului s-au dezvoltat extrem de multe limbaje de programare, dar puține dintre acestea au reuşit să se impună în timp şi ca arie de utilizare. Să menționăm câteva repere istorice în evoluția limbajelor de programare:

- 1955 – limbajul **FORTRAN** (**FOR**mula **TRAN**slation), destinat aplicațiilor tehnico-ştiințifice cu caracter numeric;

- 1960 – limbajul ALGOL (ALGOrithmic Language), primul limbaj definit riguros, cu o sintaxă complet formalizată; conceptele introduse de colectivul coordonat de *Peter Naur* sunt utilizate şi azi de proiectanţii de limbaje de programare;
- 1960 – prima versiune a limbajului COBOL (COmmon Business Oriented Language), destinat aplicaţiilor economice;
- 1971 – *Niklaus Wirth* a conceput un limbaj care să-i ajute pe studenţi să-şi însuşească rapid şi mai ales corect principiile „artei programării"; în onoarea matematicianului francez *Blaise Pascal*, el a numit acest limbaj PASCAL;
- 1972 – *Brian Kernighan* şi *Dennis Ritchie* au conceput un limbaj cu destinaţie universală, denumit C;
- 1980 – *Bjarne Stroustrup* publică specificaţiile limbajului C++, o extensie a limbajului C destinată programării orientate pe obiect;
- 1995 – *James Gostling* publică specificaţiile limbajului *Java*, un limbaj orientat pe obiect, cu sintaxă şi principii asemănătoare cu ale limbajului C++. Java are ca prim obiectiv portabilitatea (el este independent de maşina pe care lucrează).

Desigur, sunt numai câteva dintre reperele istorice ale dezvoltării limbajelor de programare. Permanent se dezvoltă limbaje de programare specifice diferitelor domenii de programare (de exemplu, limbaje destinate inteligenţei artificiale, limbaje grafice, limbaje de programare pentru Internet etc.).

În această carte vom studia limbajul de programare C, precizând eventualele diferenţe faţă de limbajul C++. Am ales acest limbaj datorită faptului că ocupă o poziţie importantă în ofertele pentru locuri de muncă, dar şi pentru că poate fi învăţat gradual de începătorii în programare. Prin intermediul limbajului C/C++, ne vom însuşi sistematic principiile programării structurate şi, ulterior, principiile programării orientate pe obiect.

Pentru a specifica faptul că un element al limbajului este disponibil atât în C, cât şi în C++, vom nota C/C++.

1.2. *Setul de caractere*

Setul de caractere utilizat pentru scrierea programelor C/C++ este setul de caractere al codului ASCII[1].

1.3. *Identificatori*

Identificatorii, întâlniţi şi sub denumirea *nume*, au rolul de a denumi elemente ale programului: constante, variabile, funcţii etc.

Din punct de vedere sintactic, un identificator este constituit dintr-o succesiune de litere, cifre sau caracterul ' ' (liniuţa de subliniere – *underscore*), primul caracter fiind obligatoriu literă sau liniuţa de subliniere.

1. Vezi anexa 1.

Exemple

Corecte:	Incorecte:	
`Program`	`Nume Prenume`	(conține caracterul spațiu)
`Nume_Prenume_elev1`	`a+b`	(conține caracterul +)
`_unu`	`2b`	(începe cu o cifră)

Observații

1. Un identificator poate avea orice lungime, dar sunt luate în considerație numai primele 31 de caractere.
2. Atenție! Limbajul C/C++ este *case-sensitive*, adică face diferența între literele mici și literele mari. Prin urmare, identificatorii `NMx` și `Nmx` sunt diferiți!
3. Este recomandat să utilizati identificatori sugestivi, astfel încât sa nu fie necesar, pe cât posibil, să apelați la declaratii sau comentarii pentru a înțelege scopul în care este folosit un identificator. Este util sa construim identificatorii prin compunerea mai multor cuvinte sau rădăcini ale acestora. În acest caz, inițiala fiecărui cuvânt care intră în compunerea identificatorului poate fi majuscula (de exemplu, `NrMaxCuvinte`, `NrTelefon` etc).

1.4. *Cuvinte rezervate*

Cuvintele rezervate, denumite uneori și *cuvinte-cheie* (*keywords*), sunt identificatori speciali, cu înțeles predefinit, care pot fi utilizați numai în construcțiile sintactice în care sunt specificați.

Exemple

`if` – cuvânt-cheie care descrie începutul unei instrucțiuni alternative;
`while` – cuvânt-cheie care descrie începutul unei instrucțiuni repetitive.

Observații

1. În limbajul C/C++, toate cuvintele rezervate[2] se scriu numai cu litere mici.
2. În programele pe care le vom prezenta în carte, vom scrie îngroșat cuvintele rezervate.

1.5. *Comentarii*

Comentariile sunt texte care vor fi ignorate de compilator[3], dar au rolul de a explicita pentru programatori anumite secvențe de program.

2. Lista completă a cuvintelor rezervate ale limbajului C/C++ se găsește în anexa 2.
3. Compilatorul este un program care are rolul de a traduce textul programului scris în limbajul de programare (numit cod sursă) în limbaj-mașină, obținând cod-obiect.

Din punct de vedere sintactic, un comentariu este:
- o succesiune de caractere încadrată între `/*` şi `*/`; aceste comentarii pot fi formate din mai multe linii;
- o succesiune de caractere care începe cu `//` şi se termină la sfârşitul liniei.

Exemple

```
/* acesta este un comentariu care poate fi scris pe mai
multe linii */
// acest comentariu se termina la sfarsitul liniei
/*Legea lui Schyer: Daca instructiunile programului si
comentariile sunt in dezacord, atunci probabil amandoua sunt
eronate. */
```

Observaţie

În activitatea de programare, documentarea programelor este esenţială. Documentarea unui produs *software* se face în două forme: documentarea externă (manual de referinţă, rapoarte, articole etc) şi documentarea internă (comentarii inserate în program). Pentru documentarea internă a programelor, este indicat ca fiecare secvenţă de program să fie însoţită de un comentariu referitor la scopul acesteia. De asemenea, tipurile de date şi variabilele folosite trebuie să fie însoţite de comentarii referitoare la semnificaţia lor. Cu cât un program este mai detaliat comentat, cu atât este mai uşor de citit, de înţeles, de modificat, de întreţinut atât de către programatorul însuşi, cât şi de colaboratorii acestuia.

1.6. *Separatori*

Separatorii au rolul de a separa unităţile sintactice.

Ca separatori „universali" se utilizează caracterele albe (spaţiu `' '`, TAB `'\t'`, sfârşit de linie[4] – *newline* – `'\n'`) şi comentariile.

Unele construcţii sintactice utilizează şi *separatori specifici* (de exemplu, într-o declaraţie de variabile, variabilele sunt separate prin caracterul virgulă `','`) sau *delimitatori* (de exemplu, caracterul `';'` delimitează sfârşitul unei instrucţiuni sau al unei declaraţii; caracterul apostrof delimitează o constantă caracter, ghilimelele delimitează constantele şir de caractere).

4. Sfârşitul de linie este marcat prin acţionarea tastei ENTER, care (dacă lucrăm sub sistemul de operare Windows) produce combinaţia de caractere CR+LF (CR – Carriage Return – poziţionare la începutul rândului, cod ASCII 13; LF – Line Feed – trecerea la rândul următor, cod ASCII 10). Dacă lucrăm sub sistemul de operare Linux, marcajului de sfârşit de linie (caracterul *newline*) îi corespunde numai codul ASCII 10.

1.7. *Structura generală a unui program C/C++*

Privit în ansamblu, un program C/C++ este constituit dintr-o succesiune de module, denumite *funcții*. Una dintre aceste funcții este funcția principală, denumită `main()`. Aceasta este o funcție specială, care trebuie să apară obligatoriu o singură dată în orice program C/C++, deoarece execuția oricărui program începe cu funcția `main()`.

Pentru început, vom elabora programe constituite numai din funcția `main()`, urmând ca ulterior să elaborăm propriile noastre funcții.

Cel mai simplu program C/C++ arată astfel:

```
void main()
{ }
```

Evident, acest program nu face nimic! Observați că el este constituit numai din funcția `main()`.

Definirea oricărei funcții este constituită din *antetul funcției* și *corpul funcției*. *Antetul* funcției conține numele funcției, tipul rezultatului pe care îl calculează funcția și o listă de parametri prin care funcția comunică cu exteriorul ei, încadrată între paranteze rotunde:

```
tip_rezultat nume(lista_parametri)
```

Observați că în cazul funcției `main()` pe care am descris-o mai sus, lista parametrilor este vidă, iar funcția nu întoarce nici un rezultat (acest lucru este indicat prin specificarea tipului `void`).

Rezultatul returnat de funcția `main()` este preluat de sistemul de operare și de obicei oferă indicații despre modul de funcționare a programului. Când execuția unui program se termină cu succes, în mod uzual, programul returnează la încheierea execuției sale valoarea `0`.

Cel mai simplu program C/C++ care nu face nimic, dar se termină cu succes, arată astfel:

```
int main()
{return 0;}
```

Instrucțiunea `return` este utilizată pentru a încheia execuția unei funcții și a returna valoarea expresiei specificate în instrucțiunea `return` ca valoare a funcției. Dacă funcția este `main()`, se încheie o dată cu funcția `main()` și execuția programului.

Formatul general al instrucțiunii `return` este:

```
return expresie;
```

În caz că apar erori pe parcursul execuției programului (de exemplu, utilizatorul programului a introdus valori negative pentru o variabilă număr natural sau un șir de caractere acolo unde ar fi trebuit un număr), programul trebuie să se termine returnând sistemului de operare un cod de eroare diferit de `0`, de obicei specific erorii care a

apărut pe parcursul execuției programului. De exemplu, dacă pentru vârsta unui pacient a fost introdus un număr care nu se încadrează în intervalul [0, 150], prin convenție programul ar putea returna codul de eroare 1.

Corpul funcției este încadrat între acolade. În corpul funcției se vor scrie declarațiile și instrucțiunile care trebuie să fie executate în această funcție. În exemplul nostru, corpul funcției este vid, deci funcția nu face nimic.

1.8. *Tipuri de date standard*

Prin *date* înțelegem, în general, tot ceea ce este prelucrat de un calculator. Fiecare dată are un anumit *tip*.

Un *tip de date* definește mulțimea valorilor pe care le pot lua datele de tipul respectiv, modul de reprezentare a acestora în memorie, precum și operațiile care se pot efectua cu datele respective.

Orice limbaj de programare dispune de un set de tipuri de date predefinite, denumite și *tipuri de date standard*[5].

Mulțimea valorilor unui anumit tip de date reprezintă *constantele* tipului respectiv.

Tipul de date `int`

Valorile datelor de tip `int` sunt numere întregi, cuprinse în intervalul `[-32768,32767]`, reprezentate în memorie pe 2 octeți[6] (*bytes*), în cod complementar[7]. Tipul de date `int` suportă modificatorii de tip `unsigned` (datele sunt numere naturale) și `long` (modifică dimensiunea reprezentării). Se obțin astfel următoarele tipuri de date întregi:

Tip	Valori	Reprezentare
`int`	`[-32768,32767]`	2 octeți, cu semn
`unsigned int`	`[0,65535]`	2 octeți, fără semn
`long int`	`[-2147483648, 2147483647]`	4 octeți, cu semn
`unsigned long int`	`[0, 4294967295]`	4 octeți, fără semn

5. Tipurile de date predefinite depind de implementarea limbajului pe care o utilizați. În carte vom prezenta implementarea Borland C++ 3.1. Dacă veți utiliza alt mediu de programare (de exemplu, GNU C pentru Linux), veți observa că există și alte tipuri de date predefinite, precum și diferențe de implementare pentru tipurile de date prezentate în carte. De exemplu, în GNU C există și tipul long long int pentru reprezentarea numerelor întregi pe 64 biți, iar tipul int este reprezentat pe 4 octeți, în loc de 2 octeți.
6. În anexa 4 găsiți informații despre organizarea logică a memoriei.
7. În anexa 5 găsiți informații despre modalitățile de reprezentare a datelor în memorie.

Atunci când un tip de date nu este precizat, implicit este considerat `int`. Prin urmare, putem specifica numai modificatorul `long`, sau `unsigned`, sau `unsigned long`, tipul fiind implicit `int`.

Constantele întregi sunt numere întregi din intervalul corespunzător tipului. Ele pot fi precizate în baza `10` (folosind notația uzuală), în baza[8] `8` (în acest caz, constanta este precedată de un `0` nesemnificativ) sau în baza `16` (caz în care constanta are prefixul `0x` sau `0X`).

Exemple

`123`	constantă zecimală de tip `int`
`-12345678`	constantă zecimală de tip `long int`
`01234`	constantă octală
`0x1a0`	constantă hexazecimală
`0XFFFF`	constantă hexazecimală

Tipul constantei este determinat implicit de valoarea ei (`int` dacă se găsește în intervalul `[-32768,32767]`, `long int` dacă depășește intervalul tipului `int`, `unsigned long int` în rest). Dacă dorim să specificăm explicit tipul unei constante, putem să utilizăm un sufix corespunzător tipului dorit:

Sufix	Tip constantă
`u, U`	`unsigned int`
`l, L`	`long int`
`ul, Ul, uL, UL`	`unsigned long int`

Exemple

`1` constanta 1, reprezentată pe 2 octeți, cu semn (`int`)
`1u` constanta 1, reprezentată pe 2 octeți, fără semn (`unsigned int`)
`1L` constanta 1, reprezentată pe 4 octeți, cu semn (`long int`)
`1ul` constanta 1, reprezentată pe 4 octeți, fără semn (`unsigned long int`)

Tipul `char`

Tipul `char` este, de asemenea, un tip întreg, care se reprezintă pe un octet, cu semn. Acest tip suportă un singur modificator – `unsigned`:

Tip	Valori	Reprezentare
`char`	`[-128,127]`	1 octet, cu semn
`unsigned char`	`[0,255]`	1 octet, fără semn

8. În anexa 3 găsiți informații despre baze de numerație și conversii între baze de numerație.

Constantele de tip `char` (`unsigned char`) pot fi numere întregi din intervalul specificat sau caracterele care au codurile ASCII în intervalul specificat. Valorile de tip `char` par a avea o natură dual – caractere ASCII şi numere întregi. Pentru calculator nu este însă nimic ambiguu, deoarece el reţine orice caracter cu ajutorul valorii numerice asociate (codul său ASCII). De exemplu, caracterul `'A'` şi constanta 65 au pentru calculator aceeaşi semnificaţie.

Caracterele grafice (coduri ASCII de la `32` la `127`) se pot specifica încadrând caracterul respectiv între apostrofuri. De exemplu, `'a'`, `'9'`, `' '`, `'*'`.

Caracterele negrafice (dar şi cele grafice, dacă dorim) se pot specifica încadrând între apostrofuri o secvenţă de evitare (secvenţă *escape*). Secvenţele *escape* sunt formate din caracterul \ (*backslash*), urmat de codul ASCII al caracterului (exprimat în baza 8 sau exprimat în baza 16, precedat de `x`).

Exemple

Secvenţă *escape*	Caracter
`'\65'`	`'5'` (codul exprimat în baza 8)
`'\x35'`	`'5'` (codul exprimat în baza 16)
`'\5'`	caracterul ♣ (codul exprimat in baza 8)
`'\356'`	caracterul ∈ (codul exprimat in baza 8)

Unele caractere negrafice, mai des utilizate, au asociate secvenţe *escape* speciale, constituite din \ (*backslash*) şi o literă sugestivă:

Exemple

Secvenţă *escape*	Caracter
`'\b'`	Caracterul *backspace* (deplasează cursorul pe ecran cu o poziţie la stânga).
`'\t'`	Caracterul tab orizontal.
`'\n'`	Caracterul *newline* (determină trecerea cursorului la linie nouă).
`'\a'`	Caracterul *alarm* (generează un sunet).
`'\\'`	Caracterul *backslash*.
`'\''`	Caracterul apostrof.
`'\"'`	Caracterul ghilimele.

Constantele şir de caractere sunt constituite dintr-o succesiune de caractere încadrată între ghilimele.

Exemple

```
"Acesta este un sir."
"Prima linie \n A doua linie."
```

Tipuri reale

Tipurile reale ale limbajului C/C++ sunt `float` şi `double`. Tipul `double` acceptă şi modificatorul `long`:

Tip	Valori	Reprezentare
`float`	$[3.4*10^{-38}, 3.4*10^{+38}] \cup [-3.4*10^{+38}, -3.4*10^{-38}]$	4 octeţi, în virgulă mobilă
`double`	$[1.7*10^{-308}, 1.7*10^{+308}] \cup [-1.7*10^{+308}, -1.7*10^{-308}]$	8 octeţi, în virgulă mobilă
`long double`	$[3.4*10^{-4932}, 1.1*10^{+4932}] \cup [-3.4*10^{+4932}, -1.1*10^{-4932}]$	10 octeţi, în virgulă mobilă

Constantele reale care se pot reprezenta în memoria calculatorului sunt numere raţionale din intervalele specificate. Constantele reale pot fi specificate în notaţia uzuală sau în format exponenţial (ştiinţific). În forma uzuală se precizează partea întreagă şi partea zecimală a numărului, separate prin marca zecimală din sistemul englezesc (caracterul punct). În format exponenţial se poate preciza, în plus, şi un exponent al lui 10, precedat de litera `e` sau `E`. Valoarea numărului se obţine înmulţind numărul cu 10 la puterea specificată.

Exemple

```
-12.3
 3.14
 1.2e-10        (în format uzual 0.00000000012)  = 1.2 ·10^-10
-2.67E+8        (în format uzual -267000000)     = -2.67·10^8
```

Tipul `void`

Tipul `void` este un tip special, pentru care mulţimea valorilor este vidă. Acest tip se utilizează atunci când este necesar să specificăm absenţa oricărei valori (de exemplu, pentru tipul funcţiilor care nu întorc nici un rezultat, cum a fost cazul funcţiei `main`).

Observaţie

În limbajul C/C++ nu există un tip de date special pentru valori logice. Valoarea `0` este asociată valorii de adevăr `fals`, orice valoare diferită de `0` fiind asociată valorii de adevăr `adevarat`.

1.9. *Variabile*

O variabilă este o dată care își poate modifica valoarea pe parcursul execuției programului.

În limbajul C/C++, înainte de a utiliza o variabilă, trebuie să o declarăm. La declarare, trebuie să specificăm numele variabilei, tipul acesteia și, eventual, o valoare inițială pe care dorim să o atribuim variabilei.

Formatul general al unei declarații de variabile este:

```
tip nume_var1[=expresie1] [, nume_var2[=expresie2]...];
```

Observații

1. Prin `tip` specificăm tipul variabilelor care se declară.
2. Prin `nume_var1`, `nume_var2`, ... specificăm numele variabilelor care se declară (acestea sunt identificatori).
3. Se pot declara simultan mai multe variabile de același tip, separând numele lor prin virgulă.
4. La declarare, putem atribui variabilei o valoare inițială, specificând după numele variabilei caracterul `'='` și o expresie de inițializare. Expresia trebuie să fie evaluabilă în momentul declarării.
5. Parantezele `[]` utilizate în descrierea formatului general au semnificația că elementul încadrat între paranteze este opțional (poate să apară sau nu într-o declarație de variabile).

Exemple

```
int a, b=3, c=2+4;
char z;
float x=b*2.5, y;
```

Am declarat trei variabile `a`, `b`, și `c` de tip `int`, o variabilă `z` de tip `char` și două variabile `x` și `y` de tip `float`. Variabilei `b` i-am atribuit valoarea inițială `3`, variabilei `c` i-am atribuit valoarea `6`, iar variabilei `x` i-am atribuit valoarea `7.5`. Variabilelor `a`, `y` și `z` nu le-am atribuit nici o valoare inițială la declarare.

Declarația unei variabile trebuie să preceadă orice referire la variabila respectivă și poate fi plasată în interiorul unei funcții (în cazul nostru, al funcției `main()`) sau în exteriorul oricărei funcții (în cazul nostru, în exteriorul funcției `main()`). Dacă declarația este plasată în interiorul unei funcții, variabila se numește *locală* funcției, altfel se numește *globală*.

Există diferențe majore între variabilele locale și cele globale, pe care le vom studia amănunțit ulterior, când vom ști să proiectăm propriile noastre funcții. La nivel de inițiere este util să știm că *variabilele globale sunt automat inițializate cu 0.* Cele locale nu sunt inițializate.

Referitor la poziția declarațiilor de variabile în cadrul unei funcții, există diferențe între limbajul C și limbajul C++. În limbajul C++ putem plasa declarații de variabile oriunde în corpul unei funcții. În limbajul C, declarațiile de variabile trebuie să fie plasate la începutul corpului funcției, înaintea oricărei instrucțiuni.

1.10. *Preprocesare*

Preprocesorul este un program lansat în execuție automat înainte de compilare[9]. El execută toate directivele preprocesor incluse în program, efectuând substituții de texte.

Toate directivele preprocesor încep cu caracterul #. De exemplu, `#include`, `#define`, `#if`, `#undef`, `#line` etc. Studiul acestor directive depășește nivelul de inițiere în programare, prin urmare, le vom discuta detaliat mai târziu. Dar pentru a elabora programe, trebuie să învățăm directivele `#include` și `#define`.

Directiva `#include` este utilizată pentru a include într-un program un fișier antet standard sau creat de utilizator.

Un *fișier antet* (*header*) conține declarațiile funcțiilor, constantelor, variabilelor și tipurilor definite într-o bibliotecă.

Fișierul antet este specific bibliotecii[10] pe care dorim să o utilizăm și are întotdeauna extensia `h`.

Pentru a include într-un program un fișier antet standard:

```
#include <nume_fisier_antet.h>
```

Exemple

```
#include <math.h>
```

Include fișierul antet al bibliotecii de funcții matematice.

```
#include <iostream.h>
```

Include un fișier antet al bibliotecii limbajului C++ cu funcții de intrare/ieșire.

```
#include <stdio.h>
```

Include un fișier antet al bibliotecii limbajului C cu funcții de intrare/ieșire.

Pentru a include un fișier antet creat de utilizator, sintaxa este:

```
#include "nume_fisier_antet.h"
```

9. Pentru descrierea etapelor de execuție a programelor, vezi anexa 6.
10. Fișierele antet sunt specifice mediului de programare pe care îl utilizați! De exemplu, dacă elaborați un program sub mediul de programare Borland C++ 3.1 și doriți să funcționeze și sub sistemul de operare Linux, cu compilatoarele GNU, trebuie să fiți foarte atenți, pentru că nu toate fișierele antet existente în Borland există și pentru GNU sau ele nu au neapărat același conținut.

Directiva `#define`, într-un format simplu, permite definirea unei constante simbolice:

```
#define identificator_constanta valoare
```

Ca efect, preprocesorul va substitui în program orice apariție a identificatorului de constantă cu valoarea acesteia.

Exemple

```
#define PI  3.1415
#define NrMaxElevi 35
```

Se recomandă utilizarea constantelor simbolice atunci când dorim să asociem o denumire mai sugestivă unei valori, ceea ce conduce la creșterea lizibilității programului. De asemenea, prin utilizarea constantelor simbolice, programul devine mai ușor de modificat. De exemplu, dacă am scris un program care să funcționeze pentru clase cu maximum 35 de elevi, atunci când numărul maxim de elevi în clasă se modifică, nu trebuie să parcurgem întreg programul și să schimbăm fiecare apariție a valorii modificate, este suficient să schimbăm valoarea constantei simbolice din definiția ei.

Observație

Fișierele antet conțin și definiții de constante simbolice. De exemplu, în fișierul antet `values.h` sunt definite constantele simbolice `MAXINT` (care are ca valoare `32767`, cel mai mare număr de tip `int`) și `MAXLONG` (care are ca valoare `2147483647`, cel mai mare număr de tip `long int`). În fișierul antet `conio.h` sunt definite constante simbolice pentru culori (de exemplu, `BLACK` cu valoarea `0`, `BLUE` cu valoarea `1`, `GREEN` cu valoarea `2`, `RED` cu valoarea `4` etc).

1.11. *Utilizarea funcțiilor din bibliotecile standard*

Setul de instrucțiuni al limbajului C/C++ este relativ restrâns, dar suficient pentru o implementare eficientă și concisă a algoritmilor.

Există unele operații care sunt frecvent utilizate (cum ar fi, de exemplu, citirea, scrierea, extragerea radicalului, calculul modulului, ștergerea ecranului etc) pentru care nu există instrucțiuni specifice. Ele pot fi implementate cu ajutorul instrucțiunilor existente în limbaj, dar... ar fi absurd să le implementăm de fiecare dată, în fiecare program pe care îl elaborăm. Din acest motiv, s-au constituit biblioteci de funcții, care conțin colecții de funcții de utilitate generală grupate pe categorii. Pentru a le utiliza, este suficient să includem în program fișierul antet al bibliotecii și să apelăm funcția care ne este necesară.

Pentru a utiliza funcțiile dintr-o bibliotecă trebuie să includem la începutul programului fișierul antet (*header*) care conține declarațiile funcțiilor din biblioteca

Pentru a apela o funcție trebuie să cunoaștem numele și formatul (prototipul) acesteia. În biblioteci există sute și sute de funcții, este improbabil să le putem cunoaște pe toate. Sigur că, după un timp, le vom reține pe cele pe care le vom utiliza frecvent. În rest, singura noastră soluție este să apelăm la *Help* (sistemul de autodocumentare al mediului de programare). În *Help* vom găsi prototipul tuturor funcțiilor din bibliotecile standard și explicații despre modul de funcționare a acestora. Prototipul unei funcții ne informează despre numele funcției, tipul valorii calculate de funcție și despre parametrii funcției. Cu alte cuvinte, are structura unui antet de funcție. Parametrii unei funcții au o semnificație similară argumentelor funcțiilor învățate la matematică. Atunci când utilizăm o funcție (în limbaj informatic, o apelăm), trebuie să specificăm valorile efective ale parametrilor pentru care apelăm funcția. Formatul unui apel de funcție este:

```
nume_functie(lista_valori_parametri)
```

Valorile parametrilor de la apel trebuie să corespundă ca număr, ordine și tip cu parametrii specificați în prototipul funcției.

Exemple

1. În fișierul antet `math.h`, care conține funcții matematice, este declarată funcția de extragere a radicalului `sqrt()`. Prototipul funcției `sqrt()` este:

```
double sqrt(double x);
```

Efect: calculează radicalul valorii `x`, transmise ca parametru. De exemplu, dacă dorim să calculăm $\sqrt{3}$, apelăm `sqrt(3)`.

2. În fișierul antet `conio.h` este declarată o funcție care are ca efect ștergerea ferestrei curente (mai exact, fereastra curentă, care implicit coincide cu întreg ecranul, este colorată în culoarea de fundal – implicit negru):

```
void clrscr(void);
```

Dacă dorim să schimbăm culoarea de fundal, putem apela funcția `textbackground()`, declarată tot în `conio.h`.

```
void textbackground(int culoare);
```

De exemplu, pentru a schimba culoarea ecranului în verde, vom apela funcția `textbackground()` pentru a stabili culoarea de fundal, apoi funcția `clrscr()` pentru a colora ecranul în culoarea de fundal:

```
textbackground(GREEN);
clrscr();
```

1.12. *Citirea/scrierea datelor*

Prin *citirea datelor* vom înțelege operația prin care una sau mai multe variabile primesc valori prin introducerea lor de la tastatură sau prin extragerea lor de pe un suport de memorare extern[11].

Prin *scrierea datelor* vom înțelege operația prin care rezultatele obținute în urma prelucrării datelor de intrare sunt fie afișate pe ecranul monitorului, fie stocate pe un suport extern de memorare.

Aceste operații sunt denumite frecvent și operații de intrare/ieșire.

În acest capitol vom prezenta doar citirea datelor de la tastatură și afișarea datelor pe ecran.

În limbajul C/C++ nu există instrucțiuni specializate pentru citirea/scrierea datelor. Aceste operații se realizează prin intermediul unor funcții existente în bibliotecile[12] standard ale limbajului.

Din exemplele precedente se poate deduce că operațiile de intrare/ieșire diferă în limbajul C++ față de limbajul C. De data aceasta diferențele sunt majore, deoarece operațiile de intrare/ieșire din C++ sunt proiectate din perspectiva programării orientate pe obiect.

1.13. *Citiri și scrieri în limbajul C++*

Conceptul central în operațiile de intrare/ieșire în limbajul C++ este fluxul de intrare/ieșire (denumirea originală fiind *stream*). Simplificând, putem privi un *stream* ca pe o succesiune de caractere. Dacă *stream*-ul este de intrare, secvența de caractere „curge" dinspre exterior (în cazul nostru, dinspre tastatură) către memoria calculatorului. Dacă *stream*-ul este de ieșire, secvența de caractere „curge" dinspre memoria calculatorului către exterior (în cazul nostru, ecranul monitorului).

În fișierul antet `iostream.h` sunt declarate două fluxuri standard: un flux de intrare de la tastatură, denumit `cin` (***console input***) și un flux de ieșire către ecran, denumit `cout` (***console output***).

Când dorim să citim date de la tastatură, le vom extrage din fluxul de intrare, folosind operatorul de extragere `'>>'`:

```
cin >> nume_variabila;
```

1. Operațiile de citire/scriere a datelor de pe/pe suport extern se vor studia în capitolul „Fișiere".
2. O bibliotecă este o colecție de funcții elaborate de firma care a dezvoltat mediul de programare pe care îl utilizăm (o vom denumi bibliotecă standard) sau de către alți programatori.

Ca urmare, se va citi de la tastatură o succesiune de caractere, care va fi convertită într-o dată de tipul variabilei specificate şi apoi atribuită variabilei.

Exemplu

```
#include <iostream.h>
int main()
{int x, y;
 cin>>x;
 cin>>y;
 return 0; }
```

Am declarat două variabile de tip `int` `x` şi `y` şi am citit de la tastatură valorile lor. Dacă presupunem că tastăm secvenţa de caractere

```
10 20<Enter>
```

în variabila `x` vom citi valoarea `10`, iar în variabila `y`, valoarea `20`.

Operatorul de extragere (îl vom numi şi operator de citire) îl putem utiliza şi înlănţuit. Mai exact, dacă dorim să citim succesiv mai multe variabile putem scrie:

```
cin >> nume_var_1 >> nume_var_2 >> ... >> nume_var_n;
```

De exemplu, am fi putut citi valorile variabilelor `x` şi `y` astfel:

```
cin >> x >> y;
```

Observaţii

1. La citirea cu ajutorul operatorului `'>>'`, valorile numerice care se citesc trebuie introduse de la tastatură, separate prin caractere albe.
2. Caracterele albe sunt ignorate de operatorul de citire `'>>'`.

Dacă dorim să scriem date pe ecran, vom utiliza operatorul `'<<'` de inserţie în fluxul de ieşire (denumit şi operator de ieşire sau de scriere):

```
cout << expresie;
```

Ca efect, se evaluează expresia, apoi valoarea ei este convertită într-o succesiune de caractere care va fi afişată pe ecran.

Şi operatorul de ieşire poate fi utilizat înlănţuit, atunci când dorim să scriem mai multe date:

```
cout << expresie_1 << expresie_2 << ... << expresie_n;
```

Exemplu

```
#include <iostream.h>
int a;
int main()
{int x;
 cout<<"x="; cin>>x;
 cout<<"Valoarea lui x este "<<x<<'\n';
 cout<<"Valoarea lui a este "<<a<<'\n';
 return 0;}
```

În primul rând va fi afişat pe ecran mesajul

```
x=
```

apoi vom tasta valoarea lui `x`, care se va citi după acţionarea tastei *Enter*. După care se va afişa mesajul `Valoarea lui x este` , urmat de valoarea citită a lui `x`. De exemplu, dacă vom tasta `5` ca valoare a lui `x`, pe ecran vom obţine:

```
x=5
Valoarea lui x este 5
Valoarea lui a este 0
```

Variabila `a` este globală, deci automat a fost iniţializată cu `0`. Observaţi că am scris şi caracterul `'\n'` (*newline*) pentru a trece pe linia următoare. Unii programatori preferă să utilizeze un simbol mai sugestiv care să determine trecerea la linie nouă. Acest simbol se numeşte *manipulator* şi este `endl` (*endline*). Deci am putea scrie în mod echivalent:

```
cout<<"Valoarea lui x este "<<x<<endl;
```

1.14. *Citiri şi scrieri în limbajul C*

Citirile şi scrierile în limbajul C se realizează prin intermediul unor funcţii specifice.

Citirea datelor cu format specificat

Funcţia `scanf()` permite citirea datelor sub controlul unui format specificat. Această funcţie este declarată în fişierul antet `stdio.h` şi are următorul format:

```
int scanf(format, adr_var1, adr_var2, ..., adr_varn);
```

Efect

Funcţia parcurge succesiunea de caractere introdusă de la tastatură şi extrage valorile care trebuie citite conform formatului specificat. Valorile citite sunt memorate în ordine în variabilele specificate prin adresa lor în lista de parametri ai funcţiei `scanf ()`. Adresa unei variabile se obţine cu ajutorul operatorului de referenţiere (&). Acest operator este unar (are un singur operand, care trebuie să fie o variabilă): `&variabila`.

Funcţia `scanf()` returnează numărul de valori citite corect. În cazul unei erori, citirea se întrerupe în poziţia în care a fost întâlnită eroarea.

Observaţie

Dacă în lista de parametri ai funcţiei `scanf()` specificaţi doar numele variabilei, nu adresa acesteia, funcţia va citi valoarea corespunzătoare acestei variabile, dar la ieşirea din funcţie valoarea citită nu este atribuită variabilei respective.

Parametrul `format` este un şir de caractere care poate conţine specificatori de format, caractere albe şi alte caractere.

Caracterele albe vor fi ignorate. Celelalte caractere (care nu fac parte dintr-un specificator de format) trebuie să fie prezente la intrare în poziţiile corespunzătoare.

Specificatorii de format au următoarea sintaxă:

```
% [*] [lg] [l|L] litera_tip
```

Observaţi că orice specificator de format începe cu caracterul `%`, conţine obligatoriu o literă ce indică tipul valorii care se citeşte şi, eventual, alte elemente opţionale. Litera ce indică tipul poate fi, de exemplu:

Literă_tip	Semnificaţie
`d`	Se citeşte un număr întreg scris în baza 10, care va fi memorat într-o variabilă de tip `int`.
`o`	Se citeşte un număr întreg scris în baza 8, care va fi memorat într-o variabilă de tip `int`.
`u`	Se citeşte un număr întreg scris în baza 10, care va fi memorat într-o variabilă de tip `unsigned int`.
`x`	Se citeşte un număr întreg scris în baza 16, care va fi memorat într-o variabilă de tip `int`.
`f`, `e` sau `g`	Se citeşte un număr real scris în baza 10, care va fi memorat într-o variabilă de tip `float`.
`c`	Se citeşte un caracter (în acest caz, caracterele albe prezente la intrare nu se ignoră).
`s`	Se citeşte un şir de caractere (şirul începe cu următorul caracter care nu este alb şi continuă până la primul caracter alb întâlnit sau până la epuizarea dimensiunii maxime `lg` din specificatorul de format).

Opţional, unele litere tip pot fi precedate de litera `l` sau de litera `L`. Literele `d`, `o` şi `x` pot fi precedate de litera `l`, caz în care valoarea citită va fi convertită la tipul `long int`. Litera `u` poate fi precedată de litera `l`, caz în care valoarea citită va fi convertită la tipul `unsigned long int`. Literele `f`, `e`, `g` pot fi precedate de litera `l` (caz în care valoarea citită este convertită la tipul `double`) sau de litera `L` (caz în care valoarea citită este convertită la tipul `long double`).

Opţional, poate fi specificată şi `lg` – lungimea maximă a zonei din care se citeşte valoarea. Mai exact, funcţia `scanf()` va citi maxim `lg` caractere, până la întâlnirea unui caracter alb sau a unui caracter neconvertibil în tipul specificat de `litera_tip`.

Caracterul opţional `'*'` specifică faptul că la intrare este prezentă o dată de tipul specificat de acest specificator de format, dar ea nu va fi atribuită nici uneia dintre variabilele specificate în lista de parametri ai funcţiei `scanf()`.

Observație

Caracterul `%` are o semnificație specială în parametrul format, el indică începutul unui specificator de format. Dacă dorim să specificăm în parametrul format că la intrare trebuie să apară caracterul `%`, vom utiliza construcția sintactică `%%`.

Exemple

1. Să considerăm următoarele declarații de variabile:

```
int a; unsigned long b; double x;
```

Să presupunem că de la tastatură sunt introduse caracterele: `312 -4.5 100000`. Apelând funcția:

```
scanf("%d %lf %lu", &a, &x, &b);
```

variabilei `a` i se atribuie valoarea `312`, variabilei `x` i se atribuie valoarea `-4.5`, iar variabilei `b` i se atribuie valoarea `100000`.

2. Să considerăm următoarea declarație de variabile:

```
char c1, c2, c3;
```

Să presupunem că de la tastatură sunt introduse caracterele: `312`. Apelând funcția:

```
scanf("%c%c%c", &c1, &c2, &c3);
```

variabilei `c1` i se atribuie caracterul `'3'`, variabilei `c2` i se atribuie caracterul `'1'`, iar variabilei `c3` i se atribuie caracterul `'2'`.

3. Să considerăm următoarele declarații de variabile:

```
char c1, c2; unsigned a;
```

Să presupunem că de la tastatură se introduce: `312 xd`. Apelând funcția:

```
scanf("%u%c%c", &a, &c1, &c2);
```

variabilei `a` i se atribuie valoarea `312`, variabilei `c1` i se atribuie caracterul `' '` (spațiu), iar variabilei `c2` i se atribuie caracterul `'x'`.

Apelând funcția:

```
scanf("%u %c %c", &a, &c1, &c2);
```

variabilei `a` i se atribuie valoarea `312`, variabilei `c1` i se atribuie caracterul `'x'`, iar variabilei `c2` i se atribuie caracterul `'d'`.

4. Să considerăm următoarea declarație de variabile:

```
int a, b, c;
```

Să presupunem că de la tastatură se introduce: `3=1+2`. Apelând funcția:

```
scanf("%d=%d+%d", &a, &b, &c);
```

variabilei `a` i se atribuie valoarea `3`, variabilei `b` i se atribuie valoarea `1`, iar variabilei `c` i se atribuie valoarea `2`.

Apelând funcția

```
scanf("%d=%*d+%d", &a, &b);
```

variabilei a i se atribuie valoarea 3, variabilei b i se atribuie valoarea 2. Valoarea 1 a fost citită, dar nu a fost atribuită nici unei variabile.

5. Să presupunem că de la tastatură se introduce o dată în formatul zz-ll-aa. Pentru a citi ziua, luna şi anul putem apela funcţia:

```
scanf("%d-%d-%d",&zi, &luna, &an);
```

Citirea caracterelor

Pentru citirea caracterelor au fost elaborate funcţii speciale. De exemplu:

1. Funcţia `getch()` este declarată în fişierul antet `conio.h`. Formatul funcţiei:

```
int getch(void);
```

Efect

Se citeşte de la tastatură un caracter. Funcţia returnează codul ASCII al caracterului citit. Caracterul tastat nu este afişat pe ecran.

2. Funcţia `getche()` este declarată în fişierul antet `conio.h`. Formatul funcţiei:

```
int getche(void);
```

Efect

Se citeşte de la tastatură un caracter. Funcţia returnează codul ASCII al caracterului citit. Caracterul tastat este afişat pe ecran.

Afişarea datelor cu format

Afişarea datelor pe ecran cu un format specificat se realizează apelând funcţia `printf()`. Această funcţie este declarată în fişierul antet `stdio.h` şi are următoarea sintaxă:

```
int printf(format, expresie1, expresie2, ..., expresien);
```

Efect

Se evaluează expresiile şi se scriu în ordine valorile acestora, în forma specificată de parametrul `format`.

În caz de succes, funcţia returnează numărul de caractere afişate.

Parametrul `format` este un şir de caractere care poate conţine specificatori de format şi, eventual, alte caractere. În parametrul format trebuie să existe câte un specificator de format pentru fiecare expresie din lista de parametri (specificatorul 1 corespunde expresiei 1, specificatorul 2, corespunde expresiei 2 etc).

Celelalte caractere din parametrul format vor fi afişate pe ecran în poziţiile corespunzătoare.

Un specificator de format are următoarea sintaxă:

```
% [ind] [lg] [.prec] [l|L] litera_tip
```

Observați că un specificator de format începe cu caracterul `%`, trebuie să conțină obligatoriu o literă care să indice tipul expresiei corespunzătoare și, eventual, alte elemente opționale. Litera care indică tipul poate fi, de exemplu:

Literă_tip	Semnificație
`d`	Expresia se convertește din tipul `int` și se afișează în baza 10.
`o`	Expresia se convertește din tipul `int` și se afișează în baza 8.
`u`	Expresia se convertește din tipul `unsigned` și se afișează în baza 10.
`x, X`	Expresia se convertește din tipul `int` și se afișează în baza 16 (literele care reprezintă 10, 11, 12, 13, 14, 15 sunt majuscule pentru `X` sau minuscule pentru `x`).
`f`	Expresia se convertește din tipul `float` și se afișează în formatul `parte_intreaga.parte_zecimala`.
`e, E`	Expresia se convertește din tipul `float` și se afișează în format exponențial (științific) `cifra.parte_zecimalae±exponent`. Pentru `E`, litera care precedă exponentul este `E`.
`g`	Se aplică una dintre conversiile corespunzătoare literei `f` sau literei `e` (cea care se reprezintă cu număr minim de caractere).
`c`	Expresia are ca valoare codul ASCII al unui caracter și se afișează caracterul corespunzător.
`s`	Expresia este un șir de caractere care va fi afișat pe ecran.

Opțional, înainte de `litera_tip` poate să apară litera `l` sau litera `L`. Dacă litera `l` apare înainte de `d`, `o`, `x`, `X` sau `u`, conversia se realizează din tipul `long int` (respectiv `unsigned long int`). Dacă litera `l` apare înainte de `f`, `e`, `E` sau `g`, conversia se realizează din tipul `double`. Litera L poate să preceadă doar literele `f`, `e`, `E` sau `g` (caz în care conversia se realizează din tipul `long double`).

Opțional, poate fi specificat `lg` – numărul minim de caractere pe care se va realiza afișarea. Dacă numărul de caractere necesar pentru a afișa valoarea expresiei depășește `lg`, se vor utiliza atâtea caractere câte sunt necesare. Dacă numărul de caractere necesare pentru a afișa valoarea expresiei este mai mic decât `lg`, se vor utiliza `lg` caractere (restul zonei pe care se realizează afișarea completându-se cu spații). Modul de aliniere implicit este la dreapta (deci completarea cu spații se realizează în stânga). Dacă se specifică indicatorul – (minus), atunci alinierea se va face la stânga (și completarea cu spații se va realiza la dreapta).

Pentru expresiile de tip real sau șir de caractere se poate specifica și precizia `prec`. Pentru valorile reale, aceasta indică numărul de cifre de la partea zecimală care vor fi afișate (implicit 6 zecimale). Dacă precizia este mai mică decât numărul de zecimale

ale numărului real afişat, atunci valoarea afişată va fi rotunjită (dacă prima zecimală care nu se afişează este mai mare sau egală cu 5, atunci ultima zecimală afişată va fi mai mare cu 1).

Pentru şiruri de caractere, precizia indică numărul de caractere din şir care se afişează (primele `prec`).

Exemple

Să considerăm următoarele declaraţii de variabile:

```
int a=-1, b=0567, d=0xf01a;
char c='x';
float x=-123.147, y=0.00008;
```

Să urmărim efectul următoarelor apeluri ale funcţiei `printf()`:

Apel	Pe ecran se afişează
`printf("a=%d sau a=%u\n", a, a);`	`a=-1 sau a=65535`

`printf("x=%f sau\n x=%e sau\n x=%g\n", x, x, x);`	`x=-123.147000 sau`
	`x=-1.231470e+02 sau`
	`x=-123.147`

`printf("y=%f sau\n y=%e sau\n y=%g\n", y, y, y);`	`y=0.000080 sau`
	`y=8.000000e-05`
	`sau y=8e-05`

`printf("b=%d sau b=%o sau b=%x \n", b, b, b);`	`b=375 sau b=567 sau b=177`
`printf("c=%c sau c=%d\n", c, c);`	`c=x sau c=120`
`printf("d=%d sau d=%x sau\n d=%u\n", d, d, d);`	`d=-4070 sau d=f01a sau` `d=61466`
`printf("x=%.2f\n", x);`	`x=-123.15`
`printf("b=%6dxb=%-6dx\n", b, b);`	`b= 375xb=375 x`

1.15. *Expresii*

O *expresie* este constituită dintr-o succesiune de operanzi, conectaţi prin operatori. Un *operand* poate fi o constantă, o variabilă, un apel de funcţie sau o expresie încadrată între paranteze rotunde.

Operatorii desemnează operaţiile care se execută asupra operanzilor şi pot fi grupaţi pe categorii, în funcţie de tipul operaţiilor desemnate. Pentru început, vom învăţa operatorii aritmetici, de incrementare şi decrementare, relaţionali, de egalitate, logici, logici pe biţi, de atribuire, condiţionali, de determinare a dimensiunii şi de conversie explicită.

Din punctul de vedere al priorității, operatorii pot fi grupați în 16 clase de prioritate, numerotate de la 1 la 16, 1 fiind prioritatea maximă. Pentru fiecare operator pe care îl vom prezenta vom indica clasa de prioritate.

Operatorii limbajului C/C++ sunt *unari* (se aplică unui singur operand) sau *binari* (necesită doi operanzi). Toți operatorii unari au clasa de prioritate 2.

Operatori aritmetici

Operatorii aritmetici desemnează operații aritmetice uzuale, cum sunt: `'*'` (înmulțirea), `'/'` (împărțirea), `'%'` (restul împărțirii întregi), `'+'` (adunarea), `'-'` (scăderea) și semnul algebric (operatorii unari `'+'`, `'-'`).

Operator	Denumire	Tip	Prioritate
`'+', '-'`	semn algebric	unari	2
`'*', '/', '%'`	multiplicativi	binari	4
`'+', '-'`	aditivi	binari	5

Observații

Operatorul `'%'` nu poate fi aplicat decât operanzilor întregi.

Operatorul `'/'` poate fi aplicat atât operanzilor întregi, cât și operanzilor reali, dar funcționează diferit pentru operanzii întregi, față de operanzii reali. Dacă cei doi operanzi sunt numere întregi, operatorul `'/'` furnizează câtul împărțirii întregi. Dacă cel puțin unul din cei doi operanzi este un număr real, operatorul `'/'` furnizează rezultatul împărțirii reale.

Exemple

Să considerăm următoarele declarații de variabile:

```
int a=3, b=5;
float x=2.5;
```

Expresie	Valoare	Observații
`b%2`	1	Restul împărțirii întregi a lui `b` la 2.
`x%2`	–	Eroare! Operatorul `'%'` se aplică numai operanzilor întregi.
`a/2`	1	Câtul împărțirii întregi a lui `a` la 2.
`x/2`	1.25	Câtul împărțirii reale a lui `x` la 2.
`(a+b)/2`	4	Media aritmetică dintre `a` și `b`.

Operatori de incrementare/decrementare

Operatorul de incrementare este `'++'`. Operatorul de decrementare este `'--'`. Sunt operatori unari care au ca efect mărirea (respectiv micșorarea) valorii operandului cu 1. Acești operatori se pot utiliza în formă prefixată (înaintea operandului), caz în care se efectuează mai întâi incrementarea/decrementarea și apoi se

utilizează valoarea operandului, sau în formă postfixată (după operand), caz în care se utilizează mai întâi valoarea operandului şi apoi se efectuează incrementarea/ decrementarea. Aceşti operatori pot fi aplicaţi numai variabilelor simple.

Exemple

Să considerăm următoarele declaraţii de variabile:

```
int a=3, b=5;
float x=2.5;
```

Expresie	Valoare	Observaţii
a++	4	Se măreşte valoarea variabilei a cu 1.
--x	1.5	Se micşorează valoarea variabilei x cu 1.
int c=--a;	2	Variabila declarată c primeşte valoarea 2.
int c=a--;	3	Variabila declarată c primeşte valoarea 3.
(a+b)++	-	Eroare! operatorii de incrementare/decrementare nu pot fi aplicaţi unei expresii.

Operatori relaţionali

Operatorii relaţionali sunt operatori binari şi desemnează relaţia de ordine în care se găsesc cei doi operanzi: '<' (mai mic), '>' (mai mare), '<=' (mai mic sau egal), '>=' (mai mare sau egal). Rezultatul aplicării unui operator relaţional este 1 dacă cei doi operanzi sunt în relaţia indicată de operator, şi 0, altfel. Grupa de prioritate a operatorilor relaţionali este 7.

Exemple

Expresie	Valoare	Observaţii
3>5	0	Operanzii nu sunt în relaţia >.
7<=3+12	1	Operanzii sunt în relaţia <=.

Operatori de egalitate

Operatorii de egalitate sunt operatori binari şi desemnează relaţia de egalitate (==) sau inegalitate (!=) în care se găsesc cei doi operanzi. Rezultatul aplicării unui operator de egalitate este 1, dacă cei doi operanzi sunt în relaţia indicată de operator, şi 0, altfel. Grupa de prioritate a operatorilor relaţionali este 8.

Exemple

Expresie	Valoare	Observaţii
3==2+1	1	Operanzii sunt în relaţia '=='.
3!=2+1	0	Operanzii nu sunt în relaţia '!= '.

Operatori logici globali

Există trei operatori logici globali:

Operator	Denumire	Tip	Prioritate
!	negația logică (not)	unar	2
&&	conjuncție logică (și)	binar	12
||	disjuncție logică (sau)	binar	13

În limbajul C/C++, valoarea logică `fals` este asociată valorii `0`, orice valoare diferită de `0` având semnificația `adevarat`. Prin urmare, efectul operatorilor logici globali, așa cum știm de la logică matematică, este:

x	0	≠0
!x	1	0

&&	0	≠0
0	0	0
≠0	0	1

||	0	≠0
0	0	1
≠0	1	1

Exemple

Expresie	Valoare
`!(x%2)`	`1`, dacă `x` este par. `0`, dacă `x` este impar.
`(x>=a)&&(x<=b)`	`1`, dacă `x` este în intervalul `[a,b]`. `0`, dacă `x` nu este în intervalul `[a,b]`.
`(x<a)||(x>b)`	`0`, dacă `x` este în intervalul `[a,b]`. `1`, dacă `x` nu este în intervalul `[a,b]`.

Operatori logici pe biți

Operatorii logici pe biți se aplică numai operanzilor întregi și au ca efect aplicarea operațiilor logice cunoscute (negație, conjuncție, disjuncție și disjuncție exclusivă) **bit cu bit**. Operatorii logici pe biți sunt:

Operator	Denumire	Tip	Prioritate
~	complementariere (negația pe biți)	unar	2
<<, >>	deplasare la stânga, deplasare la dreapta	binari	6
&	conjuncție logică pe biți	binar	9
^	disjuncție exclusivă pe biți	binar	10
|	disjuncție logică pe biți	binar	11

Efectul operatorilor pe biți este:

x	0	1
~x	1	0

&	0	1
0	0	0
1	0	1

^	0	1
0	0	1
1	1	0

|	0	1
0	0	1
1	1	1

Operatorii de deplasare au ca efect deplasarea reprezentării binare a primului operand spre stânga (<<) sau spre dreapta (>>). Numărul de poziții care se deplasează este specificat de cel de-al doilea operand. La deplasarea la stânga, pozițiile rămase libere în dreapta se completează cu 0. La deplasarea la dreapta, pozițiile rămase libere în stânga se completează cu 0 (dacă operandul stâng este un întreg pozitiv) sau cu 1 (dacă operandul este întreg negativ).

Exemple

Să considerăm următoarele declarații de variabile:

```
int n=3, a=0X1F8A, b=0XF0F5;
```

Pentru a determina cu ușurință reprezentările în memorie ale variabilelor a și b, am exprimat valorile lor în hexazecimal. Reprezentarea în memorie a acestor variabile este:

a=	0	0	0	1	1	1	1	1	1	0	0	0	1	0	1	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

b=	1	1	1	1	0	0	0	0	1	1	1	1	0	1	0	1
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

~a=	1	1	1	0	0	0	0	0	0	1	1	1	0	1	0	1
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

a&b=	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

a^b=	1	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

a|b=	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

a>>3=	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	1
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

a<<3=	1	1	1	1	1	1	0	0	0	1	0	1	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Dacă a ar fi fost declarat de tip unsigned, prin deplasare la dreapta, s-ar obține același rezultat, deoarece valoarea lui a este pozitivă (bitul semn este 0). Valoarea lui b este negativă (bitul 15 – bitul semn – este 1), prin deplasare la dreapta se propagă semnul, deci se completează cu 1.

b>>3=	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Observații

1. Expresia x<<n are ca efect înmulțirea operandului x cu 2^n. Expresia x>>n are ca efect împărțirea întreagă a operandului x cu 2^n.
2. Operațiile care se efectuează pe biți (deci acționează direct asupra reprezentării interne a operanzilor) sunt foarte performante (se execută foarte rapid).

Operatori de atribuire

Operatorii de atribuire sunt operatori binari care permit modificarea valorii unei variabile. Există un operator de atribuire simplu (=) și 10 operatori de atribuire compuși cu ajutorul operatorului '=' și al unui alt operator binar (aritmetic sau logic pe biți), după cum urmează:

```
variabila = expresie
```

Efect: se evaluează expresia, apoi se atribuie variabilei valoarea expresiei. Valoarea expresiei de atribuire este egală cu valoarea atribuită variabilei.

```
variabila operator= expresie
```

unde operatorul poate fi din mulțimea {*, /, %, +, -, <<, >>, &, |, ^}.

Efect: echivalent cu al unei atribuiri de tipul

```
variabila = variabila operator expresie
```

Grupa de prioritate a operatorilor de atribuire este 15.

Observație

Expresia poate fi la rândul ei o expresie de atribuire, cu alte cuvinte, operatorii de atribuire se pot utiliza înlănțuit:

```
variabila_1=variabila_2=...=variabila_n=expresie
```

Exemple

Să considerăm următoarele declarații de variabile:

```
int a=10, b;
long c=100001;
float x, y=5;
```

Expresie	Valoare	Observații
a+=2	12	La valoarea variabilei a se adună 2.
x=(a+c)/2	50005	Variabilei x i se atribuie câtul împărțirii întregi a lui a+c la 2.
b=a*=2	20	Valoarea variabilei a se înmulțește cu 2, apoi se atribuie această valoare și variabilei b.
a=y/2	2	Variabila a primește partea întreagă a valorii de tip float y/2.

Operatori condiționali

Operatorii condiționali sunt '?' și ':' și se utilizează numai împreună. Formatul unei expresii condiționale este:

```
expresie_1 ? expresie_2 : expresie_3
```

Efect: se evaluează expresie_1. Dacă expresie_1 are o valoare nenulă, atunci valoarea expresiei condiționale este egală cu valoarea expresie_2. Dacă expresie_1 are valoarea 0, atunci valoarea expresiei condiționale este egală cu valoarea expresie_3. Grupa de prioritate a operatorilor condiționali este 14.

Exemple

Expresie	Valoare
x>y ? x : y	maximul dintre x și y
x>=0 ? x : -x	modulul lui x

Operatorul de referențiere (adresă)

Este un operator unar care permite determinarea adresei zonei de memorie în care este stocată o variabilă:

```
&variabila
```

Operatorul de conversie explicită

Este un operator unar care permite conversia explicită (forțată) a tipului unei expresii la un tip specificat:

```
(tip) expresie
```

Exemple

Să considerăm următoarele declarații de variabile:

```
int a=10, b=15;
long x;
```

Expresie	Valoare	Observații
`((float)a+b)/2`	`12.5`	Media aritmetică a variabilelor `a` și `b` (în absența conversiei de tip, se calcula câtul împărțirii întregi a lui `a+b` la `2`, deci `12`).
`x=(long)a*10000`	`100000`	În absența conversiei de tip se calcula valoarea expresiei `a*10000` ca valoare de tip `int`; cum `100000` depășește `32767`, s-ar fi obținut valoarea eronată `-31072`.

Operatorul de determinare a dimensiunii

Este un operator unar care determină dimensiunea exprimată în număr de octeți a zonei de memorie necesare pentru stocarea expresiei specificate ca operand sau a unei date de tipul specificat ca operand:

```
sizeof (expresie)
sizeof (tip)
```

Exemple

Expresie	Valoare	Observații
`int v;` `sizeof(v)`	2	O variabilă de tip `int` se memorează pe 2 octeți.
`sizeof(float)`	4	O dată de tip `float` se memorează pe 4 octeți.

Operatorul virgulă

Operatorul virgulă permite compunerea mai multor expresii, astfel încât să fie tratate din punct de vedere sintactic ca o singură expresie.

```
expresie_1, expresie_2, ..., expresie_n
```

Se evaluează în ordinea de la stânga la dreapta cele `n` expresii, valoarea întregii expresii fiind egală cu valoarea `expresie_n`.

Utilizarea expresiilor compuse cu operatorul virgulă este necesară atunci când sintaxa permite evaluarea unei singure expresii (de exemplu, în instrucțiunea `for`, pe care o vom învăța în capitolul următor), dar este necesar să fie evaluate mai multe expresii. Prioritatea operatorului virgulă este minimă (16).

Exemplu

Expresie	Valoare	Observații
`int i, a, b;` `i=0,b=i+2,a=b*2`	4	Variabila `i` primește valoarea `0`, variabilei `b` i se atribuie valoarea `2`, apoi variabilei `a` i se atribuie valoarea `4`.

Evaluarea expresiilor

Evaluarea unei expresii presupune calculul valorii expresiei, prin înlocuirea în expresie a fiecărei variabile cu valoarea ei şi a fiecărei funcţii cu valoarea returnată de funcţia respectivă şi efectuarea operaţiilor specificate de operatori. În timpul evaluării expresiei se ţine cont de existenţa parantezelor, de asociativitate şi de prioritatea operatorilor:

- se evaluează în primul rând expresiile din paranteze, începând cu parantezele cele mai interioare;
- în cadrul unei expresii fără paranteze, se efectuează operaţiile în ordinea priorităţii operatorilor;
- dacă într-o expresie apare o succesiune de operatori cu priorităţi egale, se ţine cont de asociativitatea operatorilor. În limbajul C/C++, operatorii se asociază de la stânga la dreapta, cu excepţia operatorilor unari, condiţionali şi de atribuire, care se asociază de la dreapta la stânga.

Dacă toţi operanzii care intervin într-o expresie au acelaşi tip, tipul expresiei coincide cu tipul operanzilor. În cazul în care operanzii nu au acelaşi tip, pe parcursul evaluării expresiei se realizează automat o serie de conversii implicite. Pe scurt, regula pe care se bazează conversiile implicite este: *operandul care are un domeniu de valori mai restrâns este convertit la tipul operandului care are mulţimea valorilor mai amplă.* De exemplu: toţi operanzii de tip `char` se convertesc la `int`; dacă unul dintre operanzi este `long double`, celălalt va fi convertit de asemenea la `long double`; dacă unul dintre operanzi este `long`, iar celălalt este de tip `unsigned` sau `int`, el va fi convertit automat la `long`. Dacă un operand este de tip `int`, iar celălalt este de tip `unsigned`, conversia se face către `unsigned`.

Exemple

Expresie	Valoare	Observaţii
`int a=3; float b=6;` `(a+b)/2`	`4.5`	Se converteşte tipul variabilei `a` la `float`.
`int a=5;` `!a%2`	`0`	Se aplică operatorul unar `'!'` variabilei `a`, apoi se execută operaţia `'%'`.

`int a=30000, b=30000;`
`long c;`
`c=a+b;`
	`-5536`	Se adună `a` cu `b`; suma depăşeşte valoarea maximă pentru tipul `int`; valoarea obţinută în urma depăşirii (-5536) este atribuită variabilei `c`.

`int a=30000, b=30000;`
`long c;`
`c=(long)a+b;`
	`60000`	Se converteşte tipul variabilei `a` la `long`, apoi se adună `a` cu `b`, obţinând un rezultat de tip `long`, care este atribuit variabilei `c`.

Tabelul priorităţii operatorilor

Grupa	Operatori	Asociativitate
2	`! ~ + - ++ -- (tip) sizeof &`	dreapta → stânga
4	`* / %`	stânga → dreapta
5	`+ -`	stânga → dreapta
6	`<< >>`	stânga → dreapta
7	`< > <= >=`	stânga → dreapta
8	`== !=`	stânga → dreapta
9	`&`	stânga → dreapta
10	`^`	stânga → dreapta
11	`|`	stânga → dreapta
12	`&&`	stânga → dreapta
13	`||`	stânga → dreapta
14	`? :`	dreapta → stânga
15	`= *= /= %= += -= &= ^= |= <<= >>=`	dreapta → stânga
16	`,`	stânga → dreapta

1.16. *Aplicaţii*

Disc

Fie `r` un număr real, citit de la tastatură, care reprezintă lungimea razei unui cerc. Să se scrie un program care să calculeze şi să afişeze aria şi perimetrul discului de rază `r`.

Soluţie

Vom citi valoarea razei în variabila `r`. Vom calcula aria discului după formula πr^2, iar perimetrul, după formula $2\pi r$. Numărul π este un număr iraţional. Calculatorul nu poate memora numere iraţionale (care au o infinitate de zecimale), ci doar aproximări ale acestora. În fişierul antet `math.h` este definită o constantă numită `M_PI` care reprezintă o aproximare a numărului iraţional π.

Prezentăm atât programul C, cât şi programul C++ (diferenţele constau, evident, în modul de citire şi afişare).

Varianta C:

```
#include <stdio.h>
#include <math.h>
int main(void)
{double r;
 scanf("%lf",&r);
 printf("Aria=%.2lf Perimetrul=%.2lf\n",M_PI*r*r,2*M_PI*r);
 return 0;}
```

Varianta C++:

```
#include <iostream.h>
#include <math.h>
int main()
{double r;
 cin>>r;
 cout<<"Aria="<<M_PI*r*r<< " Perimetrul="<< 2*M_PI*r;
 return 0; }
```

Compus chimic

Un grup de cercetători studiază un compus chimic descoperit pe planeta Marte. În urma analizelor efectuate, au dedus că o moleculă din acest compus este formată din nC atomi de carbon, nO atomi de oxigen şi nH atomi de hidrogen. Ştiind că masa atomului de carbon este 12, masa atomului de oxigen este 16, iar masa atomului de hidrogen este 1, să se scrie un program care să calculeze şi să afişeze masa moleculară a acestui compus.

Soluţie

Varianta C

```
#include <stdio.h>
int main(void)
{ unsigned nC, nO, nH, m;
  scanf("%u%u%u",&nC,&nO,&nH);
  printf("Masa moleculara a compusului este %u \n",
          12*nC+16*nO+nH);
  return 0; }
```

Varianta C++

```
#include <iostream.h>
int main()
{ unsigned nC, nO, nH, m;
  cin>>nC>>nO>>nH;
  cout<<"Masa moleculara a compusului este " <<
        12*nC+16*nO+nH << endl;
  return 0; }
```

Triunghi

Fie x un număr natural format din 5 cifre ($x_4x_3x_2x_1x_0$). Să se afişeze un triunghi format din cifrele numărului x astfel:

- pe prima linie (în vârful triunghiului) se va afla cifra din mijloc (x_2);
- pe linia a doua se vor afla cifrele $x_3x_2x_1$;
- pe a treia linie se vor afla toate cifrele lui x.

De exemplu, dacă x=15289, triunghiul va arăta astfel:

```
  2
 528
15289
```

Soluţie

Problema constă în „spargerea" numărului x în cifre. În acest scop, am numerotat cifrele numărului x de la dreapta la stânga începând cu 0, astfel încât numărul cifrei să corespundă puterii corespunzătoare bazei (în cazul nostru, baza 10): x_0 este cifra unităţilor (deci corespunde lui 10^0), x_1 este cifra zecilor (deci corespunde puterii 10^1), x_2 este cifra sutelor (corespunde lui 10^2) ş.a.m.d.

Devine astfel evident că, pentru a extrage cifrele numărului x, trebuie să efectuăm împărţiri la 10. Pentru a obţine ultima cifră din x, vom împărţi pe x la 10 şi vom reţine restul în x_0. Eliminăm apoi ultima cifră din x (împărţind pe x la 10, x devine $x_4x_3x_2x_1$). Acum x_1 a devenit cifra unităţilor şi continuăm extragerea cifrelor numărului x în acelaşi mod. Programul în limbajul C este:

```
#include <stdio.h>
int main(void)
{unsigned long x;
 unsigned x0, x1, x2, x3;
 scanf("%lu",&x);
 x0=x%10; /* retin cifra unitatilor */
 x=x/10;  /* elimin cifra unitatilor */
 x1=x%10; /* retin cifra zecilor */
 x=x/10;  /* elimin cifra zecilor */
 x2=x%10; /* retin cifra sutelor */
 x=x/10;  /* elimin cifra sutelor */
 x3=x%10; /* retin cifra miilor */
 x=x/10;  /* in x ramane cifra zecilor de mii */
 printf("  %u\n", x2);
 printf(" %u%u%u\n", x3,x2,x1);
 printf("%lu%u%u%u%u\n",x,x3,x2,x1,x0);
 return 0; }
```

Observaţie

Variabila x în care citim numărul dat este de tip `unsigned long`, deoarece un număr de 5 cifre poate depăşi 65535.

Exerciţiu

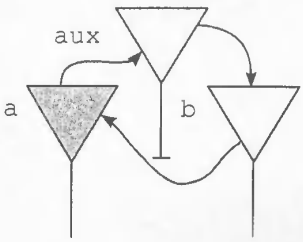
Modificaţi algoritmul precedent astfel încât să afişeze un triunghi format din cifrele unui număr de 6 cifre. De exemplu, pentru `123456`, algoritmul va afişa:

```
  34
 2345
123456
```

Schimb

Se introduc de la tastatură numerele reale `a` şi `b`. Să se interschimbe valorile variabilelor `a` şi `b`, apoi să se afişeze.

Soluţie



Să ne imaginăm că variabila `a` este un pahar cu vin alb, iar variabila `b` un pahar cu vin roşu. Trebuie să schimbăm conţinuturile celor două pahare. Singura soluţie fără „pierderi" este de a utiliza un pahar auxiliar (variabila `aux`). Vom turna vinul alb în paharul auxiliar (atribuim variabilei `aux` valoarea variabilei `a`). Acum paharul de vin alb este gol, turnăm în el vinul roşu (atribuim variabilei `a` valoarea variabilei `b`). Paharul de vin roşu (variabila `b`) a devenit disponibil, turnăm în el vinul alb din paharul auxiliar (atribuim variabilei `b` valoarea variabilei `aux`).

Această metodă este denumită sugestiv *regula celor trei pahare*, deşi analogia nu este absolut perfectă (când atribuim unei variabile valoarea altei variabile, această valoare se va găsi după atribuire în ambele, în timp ce atunci când turnăm conţinutul unui pahar de vin în alt pahar de vin...).

```
#include <stdio.h>
int main(void)
{
 float a, b, aux;
 scanf("%f%f", &a, &b);          /* 1 */
 aux=a;                          /* 2 */
 a=b;                            /* 3 */
 b=aux;                          /* 4 */
 printf("a=%f b=%f\n",a,b);      /* 5 */
 return 0;
}
```

Pentru a înţelege mai bine acest algoritm, să urmărim execuţia lui pentru un set particular de date de intrare. În acest scop, am numerotat instrucţiunile de la 1 la 5. De exemplu, să presupunem că de la tastatură se introduc valorile 3 şi 7.

Linia	a	b	aux	Explicație
1	3	7	?	Se citesc de la tastatură valorile 3 și 7 și se atribuie în ordine variabilelor a și b. Variabila aux are deocamdată o valoare necunoscută.
2	3	7	3	Se copiază valoarea variabilei a în variabila aux.
3	7	7	3	Variabilei a i se atribuie valoarea variabilei b.
4	7	3	3	Variabilei b i se atribuie valoarea variabilei aux (în care am „salvat" valoarea inițială a variabilei a).
5	7	3	3	Scriem valorile variabilelor a și b însoțite de mesaje explicative (pe ecran va apărea a=7 b=3).

Leul și iepurașul

Un iepuraș zglobiu ieși din pădure și începu să alerge pe câmpie cu o viteză constantă de v1 m/s. După un timp t0, apare la marginea pădurii un leu. Leul zări iepurașul și începu să alerge după el cu o viteză constantă de v2 m/s. Scrieți un program care să afișeze după câte secunde prinde leul iepurele sau valoarea -1 dacă leul nu prinde iepurele.

Soluție

Evident, dacă viteza leului este mai mică sau egală cu viteza iepurașului, leul nu va prinde iepurele. În caz contrar, să notăm cu x distanța de la marginea pădurii până în punctul în care leul prinde iepurele și cu t timpul după care prinde leul iepurele. Leul străbate distanța x în t secunde (fiindcă viteza leului este v2 m/s, deducem că x=v2*t). Iepurele străbate distanța x în t+t0 secunde (fiindcă viteza iepurelui este v1 m/s, deducem că x=v1*(t+t0)). Din aceste două ecuații cu două necunoscute putem afla t. Programul în limbajul C++ este:

```
#include <iostream.h>
int main()
{ float v1, v2, t;
  cout << "Viteza iepurelui= ";              cin >> v1;
  cout << "Viteza leului= ";                 cin >> v2;
  cout << "Timpul dupa care apare leul= "; cin >> t0;
  cout << (v2>v1?v1*t0/(v2-v1):-1)<<endl;  return 0;}
```

Dacă am executa programul pe următorul exemplu:

```
Viteza iepurelui= 4
Viteza leului= 5
Timpul dupa care apare leul= 2
```

pe ecran se va afișa:

```
8
```

Bazin

Un bazin se umple cu apă cu ajutorul a două robinete. Dacă lăsăm primul robinet deschis timp de `h1` ore şi al doilea timp de `h2` ore, în bazin vor fi `x` litri de apă.

Dacă lăsăm primul robinet deschis timp de `h1+1` ore şi al doilea timp de `h2-1` ore, în bazin vor fi `y` litri de apă. Scrieţi un program care să determine câţi litri de apă curg prin fiecare robinet într-o oră.

Soluţie

Să notăm cu `a` cantitatea de apă care curge într-o oră prin primul robinet şi cu `b` cantitatea de apă care curge într-o oră prin cel de-al doilea robinet. Deducem relaţiile: `x=h1*a+h2*b` şi `y=(h1+1)*a+(h2-1)*b`.

Dacă scădem cele două relaţii, obţinem `y-x=a-b`, deci `a=(y-x)+b`. Înlocuim `a` în prima relaţie şi obţinem `x=h1*(y-x)+h1*b+h2*b`, deci `b=(x-h1*(y-x))/(h2+h1)`.

```
#include <iostream.h>
int main()
{int x, y, h1, h2, b;
 cout<<"h1="; cin>>h1;
 cout<<"h2="; cin>>h2;
 cout<<"x="; cin>>x;
 cout<<"y="; cin>>y;
 cout<< "b="<<(b=(x-h1*(y-x))/(h2+h1))<<endl;
 cout<<"a= "<<(y-x)+b<<endl; return 0;}
```

De exemplu, pentru următoarele date de intrare:

```
h1=4
h2=7
x=47
y=45
```

programul va afişa:

```
b=5
a=3
```

1.17. *Probleme propuse*

1. Ce va afişa următorul program, dacă se citesc valorile `17` şi `8`?

```
#include <stdio.h>
int main(void)
{
 unsigned a, b;
 scanf("%u %u", &a, &b);
 a=a+b; b=a-b; a=a-b;
 printf("a=%u b=%u\n", a, b);
 return 0;
}
```

2. Ce va afişa următorul program, dacă se citeşte valoarea 1234?

```
#include <stdio.h>
int main(void)
{unsigned a, b, c;
 scanf("%u", &a);
 b=a%100; a=a/100; c=b*100+a;
 printf("c=%u\n", c);
 return 0; }
```

3. Ce valoare va avea variabila a la sfârşitul executării următoarei secvenţe de program?

```
int a=13, b=4;
b=a+b/2; a=a-b/2*a;
```

4. Fie a, b, c şi d patru variabile reale. Care dintre următoarele instrucţiuni atribuie variabilei d media aritmetică a valorilor variabilelor a, b şi c?

a. `d=(a+b+c)/2;`
b. `d=a/3+b/3+c/3;`
c. `d=a+b+c/3;`
d. `d=(a+b+c)/4-1;`

5. Care dintre următoarele declaraţii de variabile sunt eronate sintactic şi de ce?

a. `float a=b=0;`
b. `double z=x/2;`
c. `char d=120;`
d. `int k;i;`
e. `long a=0, b=a+3;`
f. `unsigned float a;`

6. Care dintre următoarele expresii sunt adevărate dacă şi numai dacă numărul întreg x este impar negativ?

(variantă Bacalaureat, 2000)

a. `(x%2==1) && (x<0)`
b. `(x%2!=0) || (x<0)`
c. `!((x%2==0) || (x>=0))`
d. `!((x%2==0) && (x>=0))`
e. `x%2=1 && x<0`

7. Care dintre următoarele expresii sunt adevărate dacă şi numai dacă valorile variabilelor x şi y sunt numere naturale consecutive?

(variantă Bacalaureat, 2000)

a. `x-y==1`
b. `(x==1) && (y==2)`
c. `(x-y==1) && (y-x==1)`
d. `y==x±1`
e. `(x-y==1) || (y-x==1)`

8. Să considerăm următoarele declaraţii de variabile:

```
int x=0XF0A8, v=0XFFFD;
unsigned y=0XF0A8, z=1;
float a=2.5, b=4;
char c='8';
```

Evaluați următoarele expresii:

a. `a+z/2`
b. `(v+z)/2`
c. `(v+1)/2`
d. `c-'0'`
e. `-1 > z`
f. `c>='0' && c <= '9'?"da":"nu"`
g. `x>>3&z<<4`
h. `y>>10|z`

9. În contextul declarațiilor de variabile de la exercițiul precedent, care sunt erorile din următoarele expresii?

a. `(a+z)%2`
b. `a+++x`
c. `b*b-4ac`
d. `x=(a>>2)`
e. `c=a?"da":"nu"`
f. `x&&=10`

10. În condițiile următoarelor declarații și inițializări de variabile:

```
unsigned char x=250, z=x+7, a='8';
```

Care este valoarea expresiei: `z|(a-'0')`?

a. Expresia nu se poate evalua, deoarece este eronată sintactic.
b. `1`
c. `265`
d. `true`
e. `0`
f. `9`

11. Se consideră `a` și `b` două variabile de tip `int`. Variabilei `a` i se atribuie valoarea `1`, iar variabilei `b`, valoarea `32767`. Care este valoarea expresiei `a+b`?

a. `32768`
b. `-1`
c. `-32768`
d. nu se poate calcula valoarea acestei expresii
e. `0`

12. Se consideră `x`, `y` și `z` trei variabile întregi. Care dintre următoarele expresii are valoarea diferită de `0` dacă și numai dacă `y=max(x,y,z)`?

a. `x>z?y>=x?1:0:y>=z?1:0`
b. `!(y<x || y<z)`
c. `!(y<x && y<z)`
d. `x>z && y>x || z>x && y>z`
e. `y>x && y>z`

13. Fie `x`, `a`, `r` trei numere naturale. Care dintre următoarele expresii au valoarea diferită de `0` dacă și numai dacă `x` este impar situat la distanță cel mult egală cu `r` față de `a`?

a. `x%2==1 && !(x>a+r && x<a-r)`
b. `x%2 && x<=a+r && x>a-r`
c. `!(x%2==0 || x>a+r || x<a-r)`
d. `x%2=1 &&a+r>=x && a-r<=x`

14. Care dintre următoarele expresii au valoarea diferită de `0` dacă și numai dacă variabilele întregi `a`, `b` și `c` conțin trei valori distincte două câte două?

a. `a!=b!=c!=a`
b. `!(a==b && b==c && a==c)`
c. `(a!=b) &&(b!=c)`
d. `!(a==b)&&!(b==c || a==c)`

15. Pentru a atribui variabilei reale `x` rezultatul expresiei $\frac{2ab - c^2}{0.25}$, unde `a`, `b` şi `c` desemnează variabile reale, se utilizează instrucţiunea:

(variantă Bacalaureat, 2001)

a. `x=(2*a*b)-(c*c)/0.25;`
b. `x=2*a*b-c*c/0.25;`
c. `x=(2*a*b)-(c*c)*4;`
d. `x=(2*a*b-c*c)*4;`

16. Fie `c` o variabilă de tip `char`. Scrieţi o expresie care, în cazul în care valoarea variabilei `c` este o literă mică, atribuie variabilei `c` majuscula corespunzătoare.

17. Fie `x` un număr natural. Scrieţi o expresie care să utilizeze operatori logici pe biţi astfel încât:

a. să înmulţească valoarea variabilei `x` cu 2^n ($0 \le n \le 15$);
b. să împartă valoarea variabilei `x` cu 2^n ($0 \le n \le 15$);
c. să aibă valoarea `1` dacă şi numai dacă `x` este impar;
d. să aibă valoarea `0` dacă şi numai dacă bitul `n` ($0 \le n \le 15$) din `x` este `0`;
e. să anuleze bitul `n` ($0 \le n \le 15$) din `x`;
f. să seteze bitul `n` ($0 \le n \le 15$) din `x` (să-i atribuie valoarea `1`).

18. Scrieţi un program care citeşte de la tastatură lungimea laturii unui triunghi echilateral şi afişează pe ecran lungimea înălţimii triunghiului şi aria sa, pe linii diferite, însoţite de mesaje explicative.

19. Fie `x1`, `x2`, `x3`, `x4`, `x5` cinci valori reale. Scrieţi un program care să folosească o singură variabilă suplimentară pentru a permuta circular valorile celor cinci variabile (adică în final `x1` să aibă valoarea iniţială a variabilei `x2`, `x2` valoarea iniţială a variabilei `x3`, `x3` valoarea iniţială a variabilei `x4`, `x4` valoarea iniţială a variabilei `x5`, iar `x5` valoarea iniţială a variabilei `x1`).

20. O broască ţestoasă parcurge o distanţă de `D` kilometri în `H` ore. Să se scrie un program care să calculeze şi să afişeze viteza cu care se deplasează broasca ţestoasă (exprimată în metri/secundă).

21. Doi colegi (Vasilică şi Ionică) pleacă simultan din oraşele în care locuiesc, unul către celălalt. Ştiind că distanţa dintre cele două oraşe este `D`, că Vasilică merge cu viteza `v1`, iar Ionică merge cu viteza `v2` (`D`, `v1`, `v2` sunt numere reale), scrieţi un program care calculează după cât timp se întâlnesc cei doi colegi şi la ce distanţă de oraşul în care locuieşte Vasilică.

22. Fie A şi B două puncte în plan, specificate prin coordonatele lor carteziene. Să se scrie un program care să calculeze şi să afişeze lungimea segmentului AB.

23. Scrieţi un program care citeşte de la tastatură un număr real `x` şi afişează pe ecran valoarea funcţiei:

$$f(x) = \begin{cases} \sqrt{|2-x^2|}, & \text{dacă } x \le 10 \\ \frac{-2x+1}{3}, & \text{dacă } x > 10 \end{cases}.$$

24. Consultați sistemul *Help* al mediului de programare pentru a afla care sunt funcțiile care permit generarea numerelor aleatoare. Aceste funcții sunt declarate în fișierul antet `stdlib.h`. Scrieți un program care să citească două numere naturale nenule `a` și `b` și care să genereze și să afișeze:

a. un număr natural aleator mai mic decât `a`;
b. un număr întreg aleator din intervalul [`-a, b`];
c. un număr natural aleator din intervalul [`a, b`];
d. un număr real aleator din intervalul [`-a, b`].

2. Instrucțiunile limbajului C/C++

2.1. *Instrucțiunea expresie*

```
expresie;
```

Efect

Se evaluează expresia.

Exemple

```
i++;        //se incrementeaza valoarea variabilei i
p*=2;       //se inmulteste cu 2 valoarea variabilei p
clrscr();   //se sterge ecranul
```

Observații

1. Ultima expresie este constituită dintr-un apel de funcție. Pentru a apela funcția standard `clrscr()`, trebuie să includem la începutul programului fișierul antet `conio.h`, în care se găsește prototipul (declarația) funcției:

```
void clrscr(void);
```

ceea ce înseamnă că funcția nu are nici un parametru și nu întoarce nici un rezultat.
2. Este permis ca expresia să fie vidă. Instrucțiunea devine `;` (instrucțiunea vidă).
3. Este permisă utilizarea oricărei expresii sintactic corecte, chiar și atunci când instrucțiunea nu generează nici un efect, de exemplu: `2+5;`
4. Orice instrucțiune din limbajul C/C++ se termină cu caracterul `';'`.

2.2. *Instrucțiunea compusă*

```
{
 declaratii
 instructiune_1
 instructiune_2
   ...
 instructiune_n
}
```

Efect

Se execută în ordine instrucțiunile specificate.

Observații

1. Utilizarea instrucțiunilor compuse este necesară atunci când sintaxa permite executarea unei singure instrucțiuni, dar este necesară efectuarea mai multor operații (de exemplu, într-o instrucțiune `if`, `for`, `while` sau `do-while`).
2. Declarațiile care apar într-o instrucțiune compusă sunt **locale** instrucțiunii. Mai exact, ele sunt valabile numai în corpul instrucțiunii compuse, din momentul declarării lor până la sfârșitul instrucțiunii.

2.3. *Instrucțiunea* `if`

```
if (expresie) instructiune_1
    else instructiune_2
```

Efect

Se evaluează expresia. Dacă valoarea expresiei este diferită de `0`, se execută `instructiune_1`, altfel se execută `instructiune_2`.

Exemplu

```
if (a) cout << a << " este nenul" << endl;
    else cout << a << " este nul" << endl;
```

Observații

1. Expresia se încadrează obligatoriu între paranteze rotunde.
2. Dacă `instructiune_2` este vidă, ramura `else` poate să lipsească, obținându-se o formă simplificată a instrucțiunii `if`:

```
if (expresie) instructiune
```

Exemplu

Să calculăm în variabila `max` maximul dintre `a` și `b`:

```
max=a;
if (max<b) max=b;
```

2.4. *Instrucțiunea* `switch`

```
switch     (expresie)
{ case     expresie-constanta1: secventa-instructiuni1
  case     expresie-constanta2: secventa-instructiuni2
    ...
  case     expresie-constantan: secventa-instructiunin
  default:                      secventa-instructiunin+1
}
```

Efect

Se evaluează expresia.

Se compară succesiv valoarea expresiei cu valorile expresiilor constante care etichetează alternativele `case`. Dacă se întâlneşte o alternativă `case` etichetată cu valoarea expresiei, se execută secvenţa de instrucţiuni corespunzătoare şi toate secvenţele de instrucţiuni care urmează, până la întâlnirea instrucţiunii `break` sau până la întâlnirea acoladei închise, care marchează sfârşitul instrucţiunii `switch`. Dacă nici una dintre valorile etichetelor alternativelor `case` nu coincide cu valoarea expresiei, se execută secvenţa de instrucţiuni de pe ramura `default`[13].

Exemplu

În funcţie de valoarea variabilei de tip `char c` (`'+'`, `'-'`, `'*'` sau `'/'`), vom efectua operaţia corespunzătoare între variabilele `x` şi `y`. Dacă variabila `c` are valoarea `'~'` sau `'!'`, vom da mesajul `"Nu e operator binar!"`, iar dacă `c` are orice altă valoare, vom da mesajul `"Eroare"`.

```
switch (c)
{
     case '*':   x *= y; break;
     case '/':   x /= y; break;
     case '+':   x += y; break;
     case '-':   x -= y; break;
     case '~':
     case '!':   cout << "Nu e operator binar!"; break;
     default :   cout << "Eroare!";
}
```

Observaţii

1. Expresia se încadrează obligatoriu între paranteze rotunde.
2. Pe fiecare alternativă `case` este permisă executarea mai multor instrucţiuni.
3. Dacă $\texttt{secventa-instructiuni}_{n+1}$ este vidă, ramura `default` poate lipsi (similar ramurii `else`).
4. Instrucţiunea `switch` este o generalizare a instrucţiunii `if`. Spre deosebire de `if`, care permite selectarea unei alternative din maximum două posibile, `switch` permite selectarea unei alternative din maximum `n+1` posibile. O altă diferenţă majoră constă în faptul că în `if` se execută instrucţiunea corespunzătoare valorii expresiei şi atât, în timp ce în `switch` se execută şi toate secvenţele de instrucţiuni ale alternativelor `case` următoare.

13. În limba engleză, *default* înseamnă lipsă. Deci este alternativa care se alege în lipsă de altceva. Noi o vom numi alternativă implicită.

2.5. *Instrucțiunea* `break`

```
break;
```

Efect

Determină ieșirea necondiționată din instrucțiunea `switch`, `while`, `for` sau `do-while` în care apare.

2.6. *Instrucțiunea* `while`

```
while (expresie)
 instructiune
```

Efect

Pas 1: se evaluează expresia;

Pas 2: dacă valoarea expresiei este `0`, se iese din instrucțiunea `while`;
dacă valoarea expresiei este diferită de `0`, se execută instrucțiunea, apoi se revine la Pas 1.

Observații

1. Instrucțiunea se execută repetat cât timp valoarea expresiei este nenulă. Pentru ca ciclul să nu fie infinit, este obligatoriu ca instrucțiunea care se execută să modifice cel puțin una dintre variabilele care intervin în expresie, astfel încât aceasta să poată lua valoarea `0`, sau să conțină o operație de ieșire necondiționată din ciclu (de exemplu, `break;`).
2. Dacă expresia are de la început valoarea `0`, instrucțiunea nu se execută nici măcar o dată.
3. Sintaxa permite executarea în `while` a unei singure instrucțiuni, prin urmare, atunci când este necesară efectuarea mai multor operații, acestea se grupează într-o singură instrucțiune compusă.

Exemplu

Să calculăm răsturnatul numărului natural `n`. Răsturnatul unui număr se obține considerând cifrele numărului în ordine inversă (de la dreapta la stânga).

```
r=0;                  // in r vom obtine rasturnatul lui n
while (n)
  { r=r*10+n%10;      //"lipesc" ultima cifra din n la r
    n/=10;            //elimin ultima cifra din n
  }
```

Să urmărim pas cu pas execuția acestei secvențe de instrucțiuni, pentru n=237:

Linia	Efect	n	r
1	Se inițializează variabila r cu 0.	237	0
2	Se testează dacă n≠0, apoi se execută instrucțiunea compusă.	237	0
3	Se adaugă la sfârșitul lui r ultima cifră din n.	237	7
4	Se elimină ultima cifră din n.	23	7
2	Se testează dacă n≠0, apoi se execută instrucțiunea compusă.	23	7
3	Se adaugă la sfârșitul lui r ultima cifră din n.	23	73
4	Se elimină ultima cifră din n.	2	73
2	Se testează dacă n≠0, apoi se execută instrucțiunea compusă.	2	73
3	Se adaugă la sfârșitul lui r ultima cifră din n.	2	732
4	Se elimină ultima cifră din n.	0	732
2	Se testează dacă n≠0; se iese din while.	0	732

2.7. *Instrucțiunea* do-while

```
do
  instructiune
while (expresie);
```

Efect

Pas 1: se execută instrucțiunea;
Pas 2: se evaluează expresia;
Pas 3: dacă valoarea expresiei este 0, se iese din instrucțiunea repetitivă;
dacă valoarea expresiei este diferită de 0, se revine la Pas 1.

Observație

Spre deosebire de while, instrucțiunea do-while execută instrucțiunea specificată cel puțin o dată, chiar dacă de la început valoarea expresiei este 0, deoarece evaluarea expresiei se face după execuția instrucțiunii.

În afară de momentul evaluării expresiei (înainte sau după executarea instrucțiunii), alte diferențe între cele două instrucțiuni repetitive nu există. Prin urmare, preferăm să utilizăm instrucțiunea while când este necesar să testăm o condiție înainte de efectuarea unor prelucrări. Preferăm să utilizăm do-while când condiția depinde de la început de prelucrările din ciclu și, prin urmare, este necesar să o testăm după executarea instrucțiunii.

Exemplu

Să numărăm cifrele numărului natural memorat în variabila n:

```
nr=0;              //in nr vom obtine numarul de cifre
do
  {n/=10;          //elimin ultima cifra din n
   nr++;}          //o numar
while (n); //actiunea se repeta cat timp n mai are cifre
```

Dacă în locul instrucțiunii `do-while` am fi utilizat o instrucțiune `while`, algoritmul nu ar fi funcționat și pentru `n=0`, deoarece testând condiția la început, nu s-ar fi executat nici o dată instrucțiunea din ciclu, deci valoarea variabilei `nr` ar fi rămas `0` (greșit! numărul `0` are o cifră!).

2.8. *Instrucțiunea* `for`

```
for (expresie_initializare; expresie_continuare; expresie_reinitializare)
 instructiune
```

Efect

Pas 1: se evaluează expresia de inițializare;
Pas 2: se evaluează expresia de continuare;
Pas 3: dacă valoarea expresiei de continuare este `0`, se iese din instrucțiunea repetitivă `for`;
dacă valoarea expresiei de continuare este diferită de 0:
– se execută instrucțiunea;
– se evaluează expresia de reinițializare;
– se revine la Pas 2.

Observații

Instrucțiunea `for` este tot o instrucțiune repetitivă, ca și `while` și `do-while`. Nu este o instrucțiune strict necesară, ea poate fi simulată cu instrucțiunea `while` astfel:

```
expresie_initializare;
while (expresie_continuare)
      {  instructiune
         expresie_reinitializare;  }
```

Totuși, majoritatea programatorilor preferă să utilizeze instrucțiunea `for`, deoarece este un instrument puternic, flexibil, care ne permite să exprimăm foarte concis prelucrări de natură repetitivă.

Exemplu

Fie `n` un număr natural (`n<10`) citit de la tastatură. Să se calculeze:

`n!=1*2*...*n` (`n` factorial). Prin definiție, `0!=1`.

Observăm că pentru orice `n>0` are loc relația:

`n!=1*2*...*(n-1)*n=(1*2*...*(n-1))*n=(n-1)!*n`

Prin urmare, pentru a calcula valoarea factorialului pentru `n` este suficient să înmulțim cu `n` valoarea factorialului pentru `n-1`. Putem calcula `n!` din aproape în aproape, utilizând instrucțiunea `while` astfel:

```
f=i=1;
while (i<=n)
      { f*=i;
        i++; }
```

Aceleași prelucrări pot fi descrise mult mai concis, utilizând instrucțiunea `for` astfel:

```
for (f=i=1; i<=n; f*=i++);
```

Observați că instrucțiunea care se execută în `for` este instrucțiunea vidă. Toate prelucrările au fost descrise cu ajutorul celor trei expresii care intervin în `for`.

Observație

Oricare dintre cele trei expresii care intervin în `for` poate să fie vidă. Dar și în acest caz, caracterul separator `';'` trebuie să apară.

Exemplu

```
for ( ; ; )
    { cin >> n;
      if (n>0) break;  }
```

Observați că toate cele trei expresii sunt vide. Instrucțiunea `for` citește în mod repetat un număr de la tastatură, până când numărul citit este pozitiv (atunci se iese din `for`, datorită instrucțiunii `break;`).

Exercițiu

Scrieți o instrucțiune `do-while` echivalentă, care să nu utilizeze `break;`.

2.9. *Aplicații*

Modul

Se introduce de la tastatură un număr întreg `x`. Scrieți un program care calculează și afișează modulul numărului `x`.

Soluție

Funcția modul este definită astfel: $|x| = \begin{cases} x, \text{ dacă } x \geq 0 \\ -x, \text{ dacă } x < 0 \end{cases}$

```
#include <stdio.h>
int main(void)
{int x, m;
 scanf("%d", &x);
 if (x<0) m=-x;
    else m=x;
 printf("Modulul este %d\n", m);
```

Observație

Observați că nu am testat decât condiția x<0. Evident, este suficient. Dacă am pus întrebarea *moneda a căzut cu stema în sus?* și ni s-a răspuns *nu!*, este inutil să mai întrebăm dacă a căzut cu cealaltă față în sus.

Paritate

Se introduce de la tastatură un număr întreg x. Scrieți un program care testează dacă x este un număr par.

Soluție

Numărul x este par dacă este divizibil cu 2 (adică restul împărțirii lui x la 2 este 0). Programul în limbajul C este:

```
#include <stdio.h>
int main(void)
{int x;
 scanf("%d", &x);
 if (x%2) printf("%d este impar\n", x);
    else  printf("%d este par\n", x);
 return 0; }
```

Ecuația de gradul al II-lea

Fie o ecuație de gradul al II-lea cu coeficienți reali $ax^2+bx+c=0$ ($a\neq0$). Scrieți un program care să rezolve ecuația în mulțimea numerelor reale.

Soluție

Se calculează mai întâi discriminantul ecuației. Dacă acesta este negativ, ecuația nu are rădăcini reale. Dacă acesta este egal cu 0, ecuația are două rădăcini reale egale (confundate). Dacă discriminantul este mai mare decât 0, ecuația are două rădăcini reale distincte.

```
#include <stdio.h>
#include <math.h>
int main(void)
{float a, b, c, d;
 scanf("%f %f %f", &a, &b, &c);
 d=b*b-4*a*c;                    /* calculam discriminantul */
 if (d<0) printf("Ecuatia nu are radacini reale\n");
     else
     if (!d) printf("x1=x2=%.2f\n", -b/(2*a));
        else   /*d este >0 */
        printf("x1=%.2f x2=%.2f\n", (-b+sqrt(d))/(2*a),
                (-b-sqrt(d))/(2*a));
 return 0; }
```

Exerciţiu

Modificaţi programul pentru cazul în care ecuaţia are gradul cel mult II (caz în care a poate fi 0).

Sumă

Orice sumă de bani S (S>7) poate fi plătită numai cu monede de 3 lei şi de 5 lei. Dat fiind S>7, scrieţi un program care să determine o modalitate de plată a sumei S numai cu monede de 3 lei şi de 5 lei.

Soluţie

Problema cere, de fapt, să găsim două numere naturale x şi y, astfel încât S să poată fi scris sub forma S=3*x+5*y.

Vom analiza următoarele trei cazuri:

1. S%3 (restul împărţirii lui S la 3) este 0. În acest caz, x=S/3, iar y=0.
2. S%3 este 1. Deoarece S>7, vom considera x=S/3-3 şi y=2, pentru că S=3*S/3+1=3*(S/3-3)+3*3+1=3*(S/3-3)+10=3*(S/3-3)+5*2.
3. S%3 este 2. În acest caz vom considera x=S/3-1 şi y=1, deoarece S=3*S/3+2=3*(S/3-1)+3+2=3*(S/3-1)+5.

```
#include <stdio.h>
int main(void)
{int S, x, y, r;
 scanf("%d", &S);
 r=S%3;
 switch (r)
        {case 0: x=S/3;   y=0; break;
         case 1: x=S/3-3; y=2; break;
         case 2: x=S/3-1; y=1;          }
 printf("3*%d+5*%d\n",x,y);
 return 0; }
```

Aria triunghiului

Se dau trei numere reale x, y, z. Să se verifice dacă aceste numere pot fi laturile unui triunghi. Dacă da, programul se va termina cu succes şi va afişa aria triunghiului. Dacă nu, programul va returna un cod de eroare specific erorii apărute.

Soluţie

Condiţiile ca x, y şi z să constituie laturile unui triunghi sunt:

a. x>0, y>0 şi z>0 (în caz contrar, vom returna codul de eroare 1);
b. x+y>z, x+z>y şi y+z>x (în caz contrar, vom returna codul de eroare 2).

Cunoscând laturile, putem calcula aria triunghiului utilizând formula lui *Heron*:

$$aria = \sqrt{p \cdot (p-x) \cdot (p-y) \cdot (p-z)},$$

unde cu p am notat semiperimetrul.

Prin urmare, este necesară utilizarea funcției radical. Această funcție este deja implementată în bibliotecile standard ale limbajului C++ și se numește `sqrt()`. Pentru a o putea utiliza, este suficient să includem în program fișierul antet `math.h`, în care se găsește declarația funcției:

```
double sqrt(double x);
```

ceea ce înseamnă că funcția are un singur parametru de tip `double` și întoarce ca rezultat o valoare de tip `double` (radicalul numărului transmis ca parametru).

```
#include <math.h>
#include <iostream.h>
int main ()
{ double x, y, z, p, aria;
  cin >> x >> y >> z;
  if (x<=0 || y<=0 || z<=0)     return 1;
  if (z≥x+y || y≥x+z || x≥y+z) return 2;
  p=(x+y+z)/2;                          //semiperimetrul
  aria=sqrt(p*(p-x)*(p-y)*(p-z));  //aria
  cout << aria << endl;
  return 0;
}
```

Secvență

Se consideră următorul program în limbajul C:

```
#include <stdio.h>
int main(void)
{int n, a, b, nr, i;
 scanf("%d", &n);              /* 1 */
 scanf("%d", &a);              /* 2 */
 i=1; nr=0;                    /* 3 */
 while (i<n)                   /* 4 */
      { scanf("%d", &b);       /* 5 */
        i++;                   /* 6 */
        if (b%a==0) nr++;      /* 7 */
        a=b;       }           /* 8 */
printf("nr=%d\n",nr);}         /* 9 */
```

a. Ce se va afișa dacă se citesc valorile 6, 4, 8, 7, 21, 3, 6?
b. Introduceți o succesiune de cel puțin trei valori, astfel încât programul să afișeze valoarea 0.
c. Care este efectul acestui program?

Soluţie

a. Se cere să verificăm efectul programului pentru un set particular de date de intrare. În acest caz, vom executa instrucţiunile algoritmului pas cu pas, urmărind valorile variabilelor care intervin în algoritm. Pentru simplitate, am numerotat liniile de la 1 la 9.

Linia	n	a	b	nr	i	Explicaţie
1	6	?	?	?	?	Se citeşte valoarea 6 şi se atribuie variabilei n. Celelalte variabile au valori necunoscute.
2	6	4	?	?	?	Se citeşte valoarea 4 şi se atribuie variabilei a.
3	6	4	?	0	1	Se iniţializează variabilele nr şi i cu valorile 0, respectiv 1.
4	6	4	?	0	1	Se testează dacă valoarea variabilei i (1) este mai mică decât valoarea variabilei n (6). Deoarece expresia i<n are valoarea adevarat, se continuă execuţia algoritmului cu instrucţiunea 5.
5	6	4	8	0	1	Se citeşte valoarea 8 şi se atribuie variabilei b.
6	6	4	8	0	2	Se măreşte valoarea variabilei i cu 1.
7	6	4	8	1	2	Testăm dacă restul împărţirii variabilei b la variabila a este 0 (adică dacă a divide b). Deoarece 4 divide 8, valoarea variabilei nr se măreşte cu 1.
8	6	8	8	1	2	Variabilei a i se atribuie valoarea variabilei b, apoi se revine la instrucţiunea de pe linia 4.
4	6	8	8	1	2	Se testează dacă valoarea variabilei i (2) este mai mică decât valoarea variabilei n (6). Deoarece expresia i<n are valoarea adevarat, se continuă execuţia algoritmului cu instrucţiunea de pe linia 5.
5	6	8	7	1	2	Se citeşte valoarea 7 şi se atribuie variabilei b.
6	6	8	7	1	3	Se măreşte valoarea variabilei i cu 1.
7	6	8	7	1	3	Testăm dacă restul împărţirii variabilei b la variabila a este 0 (adică dacă a divide b). Deoarece 8 nu divide 7, valoarea variabilei nr nu se măreşte cu 1.
8	6	7	7	1	3	Variabilei a i se atribuie valoarea variabilei b, apoi se revine la instrucţiunea de pe linia 4.
4	6	7	7	1	3	Se testează dacă valoarea variabilei i (3) este mai mică decât valoarea variabilei n (6). Deoarece expresia i<n are valoarea adevarat, se continuă execuţia algoritmului cu instrucţiunea de pe linia 5.
5	6	7	21	1	3	Se citeşte valoarea 21 şi se atribuie variabilei b.
6	6	7	21	1	4	Se măreşte valoarea variabilei i cu 1.
7	6	7	21	2	4	Testăm dacă restul împărţirii variabilei b la variabila a este 0 (adică dacă a divide b). Deoarece 7 divide 21, valoarea variabilei nr se măreste cu 1.

8	6	21	21	2	4	Variabilei a i se atribuie valoarea variabilei b, apoi se revine la instrucțiunea de pe linia 4.
4	6	21	21	2	4	Se testează dacă valoarea variabilei i (4) este mai mică decât valoarea variabilei n (6). Deoarece expresia i<n are valoarea adevarat, se continuă execuția algoritmpului cu instrucțiunea de pe linia 5.
5	6	21	3	2	4	Se citește valoarea 3 și se atribuie variabilei b.
6	6	21	3	2	5	Se mărește valoarea variabilei i cu 1.
7	6	21	3	2	5	Testăm dacă restul împărțirii variabilei b la variabila a este 0 (adică dacă a divide b). Deoarece 21 nu divide 3, valoarea variabilei nr nu se mărește cu 1.
8	6	3	3	2	5	Variabilei a i se atribuie valoarea variabilei b, apoi se revine la instrucțiunea de pe linia 4.
4	6	3	3	2	5	Se testează dacă valoarea variabilei i (5) este mai mică decât valoarea variabilei n (6). Deoarece expresia i<n are valoarea adevarat, se continuă execuția algoritmului cu instrucțiunea de pe linia 5.
5	6	3	6	2	5	Se citește valoarea 6 și se atribuie variabilei b.
6	6	3	6	2	6	Se mărește valoarea variabilei i cu 1.
7	6	3	6	3	6	Testăm dacă restul împărțirii variabilei b la variabila a este 0 (adică dacă a divide b). Deoarece 3 divide 6, valoarea variabilei nr se mărește cu 1.
8	6	6	6	3	6	Variabilei a i se atribuie valoarea variabilei b, apoi se revine la instrucțiunea de pe linia 4.
4	6	6	6	3	6	Se testează dacă valoarea variabilei i (6) este mai mică decât valoarea variabilei n (6). Deoarece expresia i<n are valoarea fals, se iese din ciclu și se continuă execuția algoritmului cu instrucțiunea de pe linia 9.
9	6	6	6	3	6	Se afișează valoarea variabilei nr (3).

b. Vom alege o secvență de exact trei valori. Observăm că prima valoare din secvență este egală cu numărul de valori care urmează, deci va trebui să fie 2. Pentru ca valoarea variabilei nr să rămână 0 (așa cum este inițial), va trebui să alegem două valori astfel încât condiția din instrucțiunea 7 să nu fie îndeplinită (adică prima valoare să nu dividă pe cea de-a doua). Prin urmare, o soluție posibilă este 2 5 7.

c. Urmărind algoritmul pas cu pas, deducem că se citește mai întâi o valoare n, apoi se citesc succesiv n numere naturale. Permanent avem memorate în variabilele a și b două numere consecutiv citite dintre cele n. La fiecare pas se testează dacă primul număr (cel memorat în variabila a) îl divide pe cel de-al doilea (cel memorat în variabila b) și dacă da, valoarea variabilei nr este incrementată (mărită cu 1). Prin urmare, algoritmul numără în variabila nr câte perechi de elemente

consecutive din secvența citită au proprietatea că primul element din pereche îl divide pe cel de-al doilea element din pereche.

Calcul

Se consideră următorul program C:

```
#include <stdio.h>
int main(void)
{int n, x, i;
 scanf("%d%d",&n,&x);          /* 1 */
 for (i=1; i<=n; i++)          /* 2 */
       x=x*x;                  /* 3 */
 printf("x=%d",x);             /* 4 */
 return 0; }
```

a. Ce se va afișa pe ecran pentru `n=3` și `x=2`?
b. Scrieți un program echivalent, care să utilizeze instrucțiunea repetitivă `while`.
c. Care este efectul acestui program?

Soluție

a. Pentru a determina efectul algoritmului pentru `n=3` și `x=2`, vom urmări pas cu pas execuția algoritmului, ilustrând în tabelul următor valorile variabilelor care intervin în acest algoritm.

Linia	n	x	i	Explicație
v	3	2	?	Se citește valoarea 3 și se atribuie variabilei n, apoi se citește valoarea 2 și se atribuie variabilei x. Valoarea variabilei i este deocamdată nedefinită.
2	3	2	1	Se inițializează variabila i cu 1. Deoarece valoarea variabilei i ≤ valoarea variabilei n, se continuă cu instrucțiunea de pe linia 3.
3	3	4	1	Se modifică valoarea variabilei x, se revine la linia 2.
2	3	4	2	Se mărește valoarea variabilei i cu 1. Deoarece valoarea variabilei i ≤ valoarea variabilei n, se continuă cu instrucțiunea de pe linia 3.
3	3	16	2	Se modifică valoarea variabilei x, se revine la linia 2.
2	3	16	3	Se mărește valoarea variabilei i cu 1. Deoarece valoarea variabilei i ≤ valoarea variabilei n, se continuă cu instrucțiunea de pe linia 3.
3	3	256	3	Se modifică valoarea variabilei x, se revine la linia 2.
2	3	256	4	Se mărește valoarea variabilei i cu 1. Deoarece valoarea variabilei i > valoarea variabilei n, se iese din ciclu și se continuă cu instrucțiunea de pe linia 4.
4	3	256	4	Se afișează valoarea 256.

b.

```
#include <stdio.h>
int main(void)
{int n, x, i;
 scanf("%d%d",&n,&x);
 i=1;
 while (i<=n)
       { x=x*x; i++;}
 printf("x=%d",x);
 return 0; }
```

Observați că înainte de a intra în ciclu am inițializat valoarea variabilei `i` cu `1`, iar în ciclu am inclus o instrucțiune care să incrementeze la fiecare pas valoarea variabilei `i`.

c. Pentru a determina efectul general al acestui program vom aplica un raționament inductiv. Deoarece valoarea variabilei x se modifică la fiecare pas, pentru a fi ușor de urmărit evoluția acestei variabile, vom nota cu `x0` valoarea inițială a variabilei `x`.

n	x	Explicație
0	$x0$	Dacă pentru `n` se citește valoarea `0`, instrucțiunea `for` nu va fi executată nici o dată, deci x rămâne cu valoarea sa citită inițial (`x0`).
1	$x0^2$	Dacă pentru `n` se citește valoarea `1`, instrucțiunea `for` va fi executată o singură dată, deci x va avea valoarea sa $x0^2$.
2	$x0^4$	Dacă pentru `n` se citește valoarea `2`, instrucțiunea `for` va fi executată de două ori; după prima execuție, x va avea valoarea sa $x0^2$, iar după cea de a doua, $x0^4$.
3	$x0^8$	Dacă pentru `n` se citește valoarea `3`, instrucțiunea `for` va fi executată de trei ori; după prima execuție, x va avea valoarea sa $x0^2$, după cea de-a doua, $x0^4$, iar după cea de-a treia, $x0^8$.

Urmărind execuția programului pentru câteva valori ale lui `n`, putem formula propoziția inductivă:

„După `n` pași, variabila x va avea valoarea $x0^{2^n}$".

Să demonstrăm propoziția: „După `n+1` pași, variabila x va avea valoarea $x0^{2^{n+1}}$".

Observăm că la pasul `n+1` variabilei `x` i se atribuie valoarea variabilei `x` de la pasul `n` înmulțită cu ea însăși, adică $x0^{2^n} * x0^{2^n} = x0^{2^n + 2^n} = x0^{2^{n+1}}$.

Numărare

Se consideră următorul program în limbajul C:

```
#include <stdio.h>
int main(void)
{int a, b, nr, i;
 scanf ("%d%d", &a, &b);        /* 1 */
 nr=0;                          /* 2 */
 for (i=a; i<=b; i++)           /* 3 */
     if (i%2==0) nr++;          /* 4 */
 printf("%d", nr);              /* 5 */
 return 0;}
```

a. Ce se va afişa pe ecran pentru a=5 şi b=8?
b. Scrieţi un program echivalent, care să utilizeze instrucţiunea do-while.
c. Care este efectul acestui program?
d. Scrieţi un program echivalent mai eficient.

Soluţie

a. Vom urmări pas cu pas execuţia acestui program:

Linia	a	b	nr	i	Explicaţie
1	5	8	?	?	Se citesc valorile 5 şi 8 şi se atribuie în ordine variabilelor a şi b. Variabilele nr şi i au deocamdată o valoare nedefinită.
2	5	8	0	?	Se iniţializează variabila nr cu 0.
3	5	8	0	5	Se iniţializează variabila i cu valoarea variabilei a (5). Deoarece valoarea variabilei i este ≤ valoarea variabilei b, se continuă cu instrucţiunea de pe linia 4.
4	5	8	0	5	Se testează dacă restul împărţirii variabilei i la 2 este 0 (adică dacă i este număr par). Deoarece i este impar, valoarea variabilei nr rămâne nemodificată. Se revine la linia 3.
3	5	8	0	6	Se măreşte valoarea variabilei i cu 1. Deoarece valoarea variabilei i este ≤ valoarea variabilei b, se continuă cu instrucţiunea de pe linia 4.
4	5	8	1	6	Se testează dacă restul împărţirii variabilei i la 2 este 0 (adică dacă i este număr par). Deoarece i este par, valoarea variabilei nr se măreşte cu 1. Se revine la linia 3.
3	5	8	1	7	Se măreşte valoarea variabilei i cu 1. Deoarece valoarea variabilei i este ≤ valoarea variabilei b, se continuă cu instrucţiunea de pe linia 4.
4	5	8	1	7	Se testează dacă restul împărţirii variabilei i la 2 este 0 (adică dacă i este număr par). Deoarece i este impar, valoarea variabilei nr rămâne nemodificată. Se revine la linia 3.

3	5	8	1	8	Se mărește valoarea variabilei i cu 1. Deoarece valoarea variabilei i este ≤ valoarea variabilei b, se continuă cu instrucțiunea de pe linia 4.
4	5	8	2	8	Se testează dacă restul împărțirii variabilei i la 2 este 0 (adică dacă i este număr par). Deoarece i este par, valoarea variabilei nr se mărește cu 1. Se revine la linia 3.
3	5	8	2	9	Se mărește valoarea variabilei i cu 1. Deoarece valoarea variabilei i este > valoarea variabilei b, se iese din instrucțiunea repetitivă și se continuă cu instrucțiunea de pe linia 5.
3	5	8	2	9	Se afișează valoarea variabilei nr (2).

b. Va trebui să simulăm instrucțiunea `for` cu ajutorul instrucțiunii `do while`:

```
#include <stdio.h>
int main(void)
{int a, b, nr, i;
 scanf ("%d%d", &a, &b);
 nr=0; i=a;
 if (i<=b)
    do
      { if (i%2==0) nr++;
        i++;}
    while (i<=b);
 printf("%d", nr);
 return 0;}
```

Observați că a fost necesar să testăm separat, înainte de a intra în instrucțiunea repetitivă `do while`, dacă `i<=b`. Dacă nu am fi pus acest test și dacă inițial am fi citit o valoare pară pentru variabila `a` strict mai mare decât valoarea pentru variabila `b`, atunci acest program ar fi afișat (incorect) valoarea `1`, în loc de `0`. Acest lucru se datorează faptului că instrucțiunea compusă subordonată instrucțiunii `do while` s-ar fi executat o dată și s-ar fi mărit valoarea variabilei `nr` cu `1`.

c. Observăm că variabila `i` primește în ordine toate valorile naturale din intervalul `[a, b]` (spunem că „variabila `i` parcurge intervalul `[a, b]`"). Pentru fiecare număr natural din intervalul `[a, b]`, se testează dacă el este par și, dacă da, se mărește valoarea variabilei `nr` cu `1`. Prin urmare, în variabila `nr` numărăm valorile pare din intervalul `[a, b]`.
d. Două programe sunt echivalente dacă pentru orice set de date de intrare produc aceleași date de ieșire.

Un program echivalent, dar mai eficient de a număra valorile pare dintr-un interval dat `[a, b]`, nu necesită parcurgerea tuturor numerelor din interval. În primul rând, observăm că în intervalul `[a, b]` sunt exact `L=b-a+1` numere naturale. În al doilea rând, observăm că aproximativ jumătate din valorile din intervalul `[a, b]` sunt pare și restul sunt impare. Am spus aproximativ jumătate deoarece trebuie să fim atenți la paritatea capetelor intervalului. Mai exact, va trebui să analizăm patru cazuri posibile:

a	b	nr	Explicație
par	impar	L/2	Se formează exact L/2 perechi de numere naturale consecutive (par, impar).
impar	par	L/2	Se formează exact L/2 perechi de numere naturale consecutive (impar, par).
par	par	1+L/2	Se formează exact L/2 perechi de numere naturale consecutive (par, impar), la care se mai adaugă b, care este, de asemenea, par.
impar	impar	L/2	Se formează exact L/2 perechi de numere naturale consecutive (impar, par), b fiind impar, nu va fi numărat.

Programul eficient de a număra valorile pare din intervalul [a, b] este:

```
#include <stdio.h>
int main(void)
{int a, b, nr, i;
 scanf ("%d%d", &a, &b);
 nr=0;
 if (a<=b)
    {nr=(b-a+1)/2;
     if (a%2==0 && b%2==0) nr++;   }
 printf("%d", nr);
 return 0;}
```

Putere

Fie n un număr natural și x un număr real. Scrieți un program care calculează pe x^n.

Soluție

$x^n = \underbrace{x*x*...*x}_{\text{de n ori}} = \underbrace{(x*x*...*x)}_{\text{de n-1 ori}} * x = x^{n-1} * x$

Evident, $x^0=1$.

Vom utiliza o variabilă p în care vom calcula puterea. La fiecare pas i (i variind de la 1 la n) vom înmulți valoarea variabilei p cu x.

```
#include <stdio.h>
int main(void)
{int n, i;
 float x, p=1;
 scanf("%d %f", &n, &x);
 for (i=1; i<=n; i++) p*=x;
 printf("%.2f", p);
 return 0;}
```

Observație
Acest program efectuează exact n înmulțiri.

Numere

Numerele întregi mai mari decât 1 sunt așezate pe 5 coloane, câte 4 pe linie, ca în exemplul următor:

2	3	4	5	
	9	8	7	6
10	11	12	13	
	17	16	15	14
...				

Coloanele sunt numerotate de la 1 la 5. Scrieți un program care să citească de la tastatură un număr natural mai mare decât 1 și care să afișeze coloana pe care este așezat numărul citit.

Soluție

Observăm că în coloanele 1, 3 și 5 sunt numai numere pare. În prima coloană sunt plasate numerele 2, 8, 18,... deci sunt numere de forma 8k+2. În coloana 5 sunt plasate numerele 6, 14, 22,... deci numere de forma 8k+6. În coloana 3 sunt plasate numerele 4, 8, 12, 16,... deci numere de forma 8k+4.

În coloanele 2 și 4 sunt plasate numere impare. Multiplii lui 8 sunt întotdeauna pe coloana 3 pe o linie pară, deci în coloana 2 pe liniile pare sunt numere de forma 8k+1. Pe coloana 2, pe liniile impare sunt numere de forma 8k+3. Prin urmare, în coloana 4, pe liniile pare sunt numere de forma 8k+7, iar pe liniile impare, numere de forma 8k+5.

```
#include <iostream.h>
int main()
{int x;
 cout<<"x="; cin>>x;
 switch (x%8)
       {
       case 1: cout<<2<<endl; break;
       case 2: cout<<1<<endl; break;
       case 3: cout<<2<<endl; break;
       case 4: cout<<3<<endl; break;
       case 5: cout<<4<<endl; break;
       case 6: cout<<5<<endl; break;
       case 7: cout<<4<<endl; break;
       case 0: cout<<3<<endl; break;
       }
return 0;}
```

Coduri şi caractere

Scrieţi un program care să afişeze toate caracterele din codul ASCII, precum şi codul lor. Se va scrie mai întâi caracterul, apoi codul ASCII corespunzător separat printr-un spaţiu, câte două grupe cod-caracter pe o linie, separate prin tab.

Soluţie

O primă idee ar fi să parcurgem succesiv toate codurile ASCII, (numere naturale de la `0` la `255`). Pentru fiecare cod, afişăm caracterul corespunzător, conform formatului specificat în problemă.

Pentru a obţine pe ecran caracterul corespunzător codului ASCII este necesar să utilizăm operatorul de forţare a tipului.

```
#include <iostream.h>
int main()
{int cod;
 for (cod=0; cod <=255; cod++)
    {cout<<(unsigned char)cod<<' '<< cod;
     cout<<'\t'; cod++;
     cout<<(unsigned char)cod<<' '<< cod;
     cout<<endl;     }
return 0; }
```

Dacă vom executa acest program, vom avea nişte surprize, datorate unora dintre primele 26 de caractere, care sunt caractere speciale. Mai exact:

Cod	Explicaţie	Corecţie
7	Nu generează nici un efect pe ecran, doar emite un sunet.	scriem `\a` în locul acestui caracter
8	Caracterul *backspace* care determină deplasarea cursorului ecran cu o poziţie la stânga.	scriem `\b` în locul acestui caracter
9	Caracterul tab orizontal determină deplasarea cursorului cu 8 poziţii la dreapta.	scriem `\t` în locul acestui caracter
10	Determină trecerea cursorului pe linia următoare în aceeaşi coloană.	scriem `\l` în locul acestui caracter
11	Caracterul tab vertical determină deplasarea cursorului în jos.	scriem `\v` în locul acestui caracter
12	Când este transmis către imprimantă se sare la începutul paginii următoare.	scriem `\f` în locul acestui caracter
13	Determină trecerea cursorului la începutul liniei curente.	scriem `\r` în locul acestui caracter
26	Reprezintă caracterul EOF (*end of file*) care reprezintă marcajul de sfârşit de fişier.	scriem `eof` în locul acestui caracter
27	Caracterul *escape*.	scriem spaţiu în locul acestui caracter

Prin urmare va trebui să tratăm separat aceste cazuri particulare.

```
#include <iostream.h>
int main()
{int cod;
 for (cod=0; cod<=255; cod++)
     {switch (cod)
           {case 7:  cout<<"\\a  " ;  break;
            case 8:  cout<<"\\b  " ;  break;
            case 9:  cout<<" \\t  " ; break;
            case 10: cout<<"\\l  " ; break;
            case 11: cout<<" \\v  "; break;
            case 12: cout<<"\\f  ";  break;
            case 13: cout<<" \\r  "; break;
            case 26: cout<<"eof ";   break;
            case 27: cout<<"     ";  break;
            default: cout<<(unsigned char)cod<<"   ";
            }
      cout<< cod;
      if (cod %2) cout<<'\n';
          else cout<<'\t';
     }
return 0;}
```

Exercițiu

Programul precedent afișează pe ecran 128 de linii, prin urmare noi vom putea analiza doar ultimele 25, celelalte defilând cu viteză. Modificați programul precedent astfel încât să pagineze rezultatele afișate (după umplerea unui ecran, să afișeze un mesaj de tipul „Apăsați o tastă pentru a continua", apoi să aștepte acționarea unei taste).

Secvență simetrică

Fie n un număr natural nenul. Să se afișeze pe ecran o secvență de lungime n, construită după procedeul ilustrat în exemplele următoare.

n=7

1	2	3	4	3	2	1

n=8

1	2	3	4	4	3	2	1

Soluție

În primul rând, observăm că secvența începe cu numerele de la 1 la n/2 în ordine crescătoare. Dacă n este impar, urmează numerele de la 1 la (n+1)/2 în ordine descrescătoare. Dacă n este par, urmează numerele de la 1 la n/2 în ordine descrescătoare.

```
#include <iostream.h>
int main()
{int n, i;
 cout<<"n= "; cin>>n;
 for (i=1; i<=n/2; i++)     cout<<i<<' ';
 for (i=(n+n%2)/2; i; i--) cout<<i<<' ';
 cout<<endl; return 0; }
```

Maxim

I. Fie a și b două numere reale, citite de la tastatură. Scrieți un program care determină și afișează pe ecran cea mai mare valoare citită.

Soluție

Compar valorile celor două variabile citite și o afișez pe cea mai mare:

```
#include <stdio.h>
int main(void)
{double a, b;
 scanf("%lf %lf", &a, &b);
 if (a<b) printf("max=%.2lf\n", b);
    else printf("max=%.2lf\n", a);
 return 0; }
```

O altă soluție, care are avantajul că o vom putea generaliza pentru oricâte elemente, este de a reține maximul într-o variabilă (să o numim max). Inițializăm variabila max cu a (inițial putem considera că a este „cel mai bun"). Apoi comparăm maximul cu valoarea variabilei b și dacă max este mai mic, îl schimbăm.

```
#include <stdio.h>
int main(void)
{double a, b, max;
 scanf("%lf %lf", &a, &b);
 max=a;
 if (max<b) max=b;
 printf("max=%.2lf\n", max);
 return 0; }
```

II. Fie a, b și c trei numere reale, citite de la tastatură. Scrieți un program care determină și afișează cea mai mare valoare citită.

Soluție

Ideea utilizată în cea de-a doua variantă a programului precedent o putem utiliza

```
#include <stdio.h>
int main(void)
{double a, b, c, max;      /* maximul dintre a, b si c */
 scanf("%lf %lf %lf", &a, &b, &c);
 max=a;                    /* initializare */
 if (max<b) max=b;         /* compar cu b */
 if (max<c) max=c;         /* compar cu c */
 printf("max=%.2lf\n", max);
 return 0; }
```

III. Se citește de la tastatură un număr natural nenul n, apoi se citesc succesiv n valori reale. Scrieți un program care determină și afișează cea mai mare valoare reală citită.

Soluție

O primă problemă este declararea datelor de intrare: se dau n valori reale, numai că de data aceasta n nu este constant, ci variabil (citit de la tastatură). Cum să declarăm un număr variabil de variabile? Nu putem și nici nu este necesar, pentru că nu este nevoie să citim fiecare număr într-o variabilă separată! Vom citi numerele succesiv în aceeași variabilă (să o numim a) și imediat ce le-am citit le vom compara cu elementul maxim.

Pentru a intui mai bine algoritmul de calcul al maximului, să ne imaginăm un șir de elevi care intră în clasă unul câte unul. Am dori ca pe măsură ce intră să reținem care este cel mai înalt. Evident, primul care intră poate fi considerat cel mai înalt (de altfel, este singurul pe care l-am văzut până acum). Când intră pe ușă un nou elev, compar înălțimea lui cu înălțimea celui pe care eu îl consider cel mai înalt de până acum. Dacă noul elev este mai înalt, reținînălțimea lui ca etalon pentru viitoarele comparații (el este cel mai înalt de până acum). Prin urmare, noi reținem permanent maximul dintre cei intrați deja.

Această idee o vom utiliza în continuare: în variabila max vom reține la fiecare moment cea mai mare valoare citită până la momentul respectiv.

```
#include <stdio.h>
int main(void)
{double a, max;  /* max - maximul dintre valorile citite */
 int n, i;
 /* i - numarul de valori citite pana la un moment dat*/
 scanf("%d", &n);
 scanf("%lf", &max);
 /* initializam max cu primul element citit */
 for (i=1; i<n; i++)
     {scanf("%lf", &a);  /* citesc o noua valoare */
      if (max<a) max=a;} /* compar cu max valoarea citita*/
 printf("max=%.2lf\n", max);
 return 0; }
```

Observaţii

1. În instrucţiunea `for` a fost necesară efectuarea mai multor operaţii (citirea unei valori şi compararea acesteia cu valoarea maximă de până acum). Prin urmare, am grupat instrucţiunile corespunzătoare într-o singură instrucţiune compusă.
2. Pentru a calcula maximul a `n` numere a fost necesară executarea instrucţiunii compuse subordonate instrucţiunii `for` de `n-1` ori.

Exerciţii

1. Rescrieţi programul precedent utilizând instrucţiunea repetitivă `while`.
2. Modificaţi programul precedent astfel încât să calculeze cel mai mic dintre elementele citite.
3. Modificaţi programul precedent astfel încât să calculeze cele mai mari două elemente dintre cele citite.

Media aritmetică

Se citeşte de la tastatură un număr natural nenul `n`, apoi se citesc succesiv `n` valori reale. Scrieţi un program care determină şi afişează media aritmetică a valorilor strict pozitive.

Soluţie

Media aritmetică se calculează împărţind suma valorilor strict pozitive la numărul lor.

În mod asemănător cu problema precedentă, vom citi succesiv cele `n` valori întregi în aceeaşi variabilă `a`. După fiecare citire, vom verifica dacă valoarea citită este strict pozitivă şi, în caz afirmativ, o vom număra şi o vom adăuga la sumă.

```
#include <stdio.h>
int main(void)
{double a, s=0;
 int n, i=0, nr=0;
 /*in i retinem cate valori am citit pana la un moment dat;
   in nr calculam cate valori pozitive am citit; in s
   calculam suma valorilor strict pozitive citite */
 scanf("%d", &n);
 while (i<n) /* cat timp nu am citit toate cele n valori */
    {scanf("%lf",&a);  /* citesc o noua valoare */
     i++;              /* numar valoarea citita */
     if (a>0)          /* valoarea este strict pozitiva */
        {s+=a;         /* o adun la suma */
         nr++;}        /* o numar */
    }
 if (nr) printf("media artimetica=%.2lf\n", s/nr);
    else printf("media aritmetica nu se poate calcula\n");
 return 0; }
```

Suma cifrelor

Se citește de la tastatură un număr natural n. Să se calculeze suma cifrelor lui n.

Soluție

Cel mai ușor se obține ultima cifră din număr (cifra unităților): n%10 (restul împărțirii lui n la 10). După ce utilizăm ultima cifră (o adăugăm la sumă), nu mai avem nevoie de ea și o eliminăm din număr (n=n/10). Cifra zecilor a devenit acum ultima cifră. Repetăm procedeul, cât timp numărul mai are cifre.

```
#include <stdio.h>
int main(void)
{int n, s=0;
 scanf("%d", &n);
 while (n)              /* cat timp mai exista cifre in n */
       { s+=n%10;       /* adun la s ultima cifra a lui n */
         n/=10; }       /* elimin ultima cifra a lui n*/
 printf("suma cifrelor=%d\n", s);
 return 0; }
```

Observație

Inițializarea unei variabile în care trebuie să calculăm o sumă se face întotdeauna cu 0 (elementul neutru la adunare). În mod similar, inițializarea unei variabile în care trebuie să calculăm un produs se face întotdeauna cu 1 (elementul neutru la înmulțire).

Exerciții

1. Urmăriți pas cu pas execuția acestui program pentru n=5672.
2. Modificați programul astfel încât să determine numărul de cifre pare ale lui n.

Perechi

Fie n un număr natural nenul. Să se genereze toate perechile (a, b), cu proprietatea că a|b, unde a și b sunt numere naturale nenule mai mici decât n.

Soluție

În primul rând, observăm că pentru ca a să dividă b trebuie ca a≤b. O primă idee ar fi ca a să parcurgă toate numerele naturale nenule mai mici decât n, apoi pentru fiecare valoare a, b să parcurgă toate numerele mai mari sau egale cu a și mai mici decât n, formând astfel toate perechile de numere (a, b) cu a≤b<n. Pentru fiecare pereche vom testa dacă a|b.

```
#include <stdio.h>
int main(void)
{int n, a, b;
 scanf("%d", &n);
 for (a=1; a<n; a++)
     for (b=a; b<n; b++)
         if (b%a==0) printf("(%d, %d) ", a, b);
 return 0; }
```

În acest program instrucțiunea `if` este executată de `n*(n-1)/2` ori, deoarece:
- când `a=1`, a doua instrucțiune `for` se execută de `n-1` ori;
- când `a=2`, a doua instrucțiune `for` se execută de `n-2` ori;
- când `a=3`, a doua instrucțiune `for` se execută de `n-3` ori;

(...)
- când `a=n-1`, a doua instrucțiune `for` se execută o dată;

În total: `1+2+3+...+n-1=n*(n-1)/2` executări ale instrucțiunii `if`.

O idee mai bună ar fi ca, având `a` fixat, să nu mai parcurgem cu variabila `b` toate numerele naturale de la `a` la `n-1`, ci doar multiplii lui `a`:

```
#include <stdio.h>
int main(void)
{int n, a, m;
 scanf("%d", &n);
 for (a=1; a<n; a++)
     for (m=1; a*m<n; m++)
         printf("(%d, %d) ", a, a*m);
 return 0; }
```

În acest mod, programul a devenit mai eficient, obținând direct perechile corecte.

Divizori

Dat fiind `n` un număr natural, să se determine toți divizorii naturali ai lui `n`.

Soluție

În primul rând, observăm că orice număr natural `n` are ca divizori pe `1` și pe `n` (aceștia se numesc divizori improprii). În afară de divizorii improprii, cel mai mic divizor ar putea fi `2`, iar cel mai mare `n/2`. Prin urmare, pentru a afla divizorii proprii ai lui `n`, vom parcurge toate numerele naturale din intervalul `[2, n/2]` și vom testa pentru fiecare număr natural `x` din acest interval dacă divide sau nu pe `n` (adică dacă restul împărțirii lui `n` la `x` este `0`).

```
#include <stdio.h>
int main(void)
{int n, x;
 scanf("%d", &n);
 printf("1 ");
 for (x=2; x<=n/2; x++)     /* parcurg intervalul [2,n/2] */
     if (n%x==0)            /* x divide n? */
         printf("%d ", x); /* x este divizor, il scriu */
 printf("%d\n", n);
 return 0; }
```

Și aici se poate da o soluție mai bună utilizând faptul că „dacă `d` este divizor al lui `n`, atunci și `n/d` este divizor a lui `n`". Procedând în acest mod, `x` va trebui să parcurgă doar intervalul $[2, \sqrt{n}]$.

```
#include <stdio.h>
#include <math.h>
int main(void)
{int n, x, rad;
 scanf("%d", &n);
 rad=sqrt(n);        /*calculam radicalul o singura data  */
 printf("1 ");
 for (x=2; x<=rad; x++)  /* parcurgem intervalul [2, √n ] */
   if (n%x==0)           /* x divide n? */
     printf("%d %d ", x, n/x);  /*x si n/x sunt divizori */
 printf("%d\n", n);
 return 0; }
```

Observăm că nu este necesar să calculăm radicalul la fiecare evaluare a expresiei din instrucțiunea `for`. Este suficient să îl calculăm o dată și să îl memorăm într-o variabilă pentru a-l utiliza ori de câte ori este necesar.

Număr prim

Fie `n` un număr natural, `n>1`, citit de la tastatură. Scrieți un program care verifică dacă numărul `n` este prim.

Soluție

Pentru a testa dacă numărul `n` este prim, vom verifica dacă are sau nu divizori proprii (divizori diferiți de `1` și de `n`). Pentru aceasta vom parcurge intervalul numerelor naturale de la `2` la $\sqrt{n}^2$, verificând pentru fiecare număr dacă este sau nu divizor al lui `n`. Pentru a reține rezultatul testului, vom utiliza o variabilă denumită `prim` cu semnificația „`prim` este `1`, dacă `n` este prim, și `0`, dacă `n` nu este prim". Inițial, variabilei `prim` îi atribuim valoarea `1` („prezumția de nevinovăție").

Pe parcursul verificării, dacă găsim un divizor, variabilei `prim` îi atribuim valoarea `0` („dovada" că `n` nu este prim a fost găsită).

Verificarea continuă cât timp avem *unde* căuta (nu am ajuns la $\sqrt{n}$) și avem *de ce* căuta (nu am găsit încă „dovada vinovăției" – un divizor al lui `n`).

```
#include <stdio.h>
#include <math.h>
int main(void)
{int n, d=2, prim=1;
 scanf("%d", &n);
 while (d<=sqrt(n) && prim)
       if (n%d==0) prim=0;       /* am gasit un divizor */
           else d++;             /* cautam mai departe */
  if (prim) printf("%d este prim", n);
     else printf("%d nu este prim", n);
 return 0; }
```

Am putea îmbunătăți performanțele acestui program observând că singurul număr prim par este 2. Vom testa în primul rând dacă n este par (caz în care, exceptându-l pe 2, nu este prim), apoi vom verifica numai numerele impare.

```
#include <stdio.h>
#include <math.h>
int main(void)
{int n, d, prim=1, rad;
 scanf("%d", &n);
 if (n>2)
    if (n%2==0) prim =0;
       else
        for (d=3, rad=sqrt(n); d<=rad && prim; d+=2)
           if (n%d==0) prim=0;     /* am gasit un divizor */
  if (prim) printf("%d este prim", n);
     else printf("%d nu este prim", n);
 return 0; }
```

Exerciții

1. Urmăriți pas cu pas execuția acestui program pentru n=13, n=8 și n=49.
2. Modificați programul astfel încât să afișeze toate numerele naturale prime ≤ n.

Descompunere în factori primi

Fie n un număr natural (n>1), citit de la tastatură. Scrieți un program care să afișeze descompunerea numărului natural n în factori primi.

De exemplu, pentru n=720, programul va afișa 2^4*3^2*5^1, ceea ce înseamnă că divizorii primi sunt 2, 3 și 5, având respectiv multiplicitățile 4, 2 și 1.

Soluție

Vom aplica algoritmul învățat la matematică: considerăm toate numerele naturale începând cu 2. Pentru fiecare număr, verificăm dacă este divizor al lui n. În caz afirmativ, calculăm multiplicitatea acestui divizor în n, împărțind succesiv pe n la divizor și calculând numărul de împărțiri efectuate, după care afișăm atât divizorul găsit, cât și multiplicitatea sa. Procedeul se repetă cât timp n mai are divizori (n>1).

```
#include <stdio.h>
int main(void)
{int n, d, m;
 scanf("%d", &n);
 for (d=2; n>1; d++)
     {/* calculez multiplicitatea lui d in n */
       for (m=0; n%d==0; m++, n/=d);
       if (m) /* d este divizor */
         printf("Divizorul %d, multiplicitatea %d\n",d,m);
     }
 return 0; }
```

Exerciții

1. Urmăriți pas cu pas execuția acestui program pentru n=2500.
2. Observați că nu am testat nicăieri dacă d este prim. Nu este necesar, deoarece datorită faptului că atunci când găsim un divizor al lui n îl împărțim pe n la divizor ori de câte ori este posibil, orice divizor nou găsit este prim. Demonstrați această afirmație prin reducere la absurd.
3. Modificați programul astfel încât să afișeze numai cel mai mare divizor prim.
4. Modificați programul astfel încât să calculeze numărul total de divizori ai lui n.

Cel mai mare divizor comun

Fie n și m două numere naturale, citite de la tastatură. Scrieți un program care să calculeze și să afișeze c.m.m.d.c. (n, m).

Soluție

Vom aplica algoritmul lui Euclid, învățat la matematică. Cât timp m≠0, împărțim pe n la m și calculăm restul; la pasul următor, împărțitorul devine deîmpărțit, iar restul devine împărțitor. Ultimul rest≠0 este c.m.m.d.c.

De exemplu, pentru a=495 și b=720, *cel mai mare divizor comun* prin algoritmul lui Euclid se obține astfel:

Deîmpărțit		Împărțitor	Rest
495	=	720 · 0	+ 495
720	=	495 · 1	+ 225
495	=	225 · 2	+ 45
225	=	45 · 5	+ 0 ⇒ c.m.m.d.c.(495,720)=45

```
#include <stdio.h>
int main(void)
{int n, m, rest;
 scanf("%d %d", &n, &m);
 while (m)
       { rest=n%m;      /* calculez restul                */
         n=m;           /* impartitorul devine deimpartit */
         m=rest; }      /* restul devine impartitor       */
 printf("cmmdc=%d", n);/* ultimul rest diferit de 0       */
 return 0; }
```

Exerciții

1. Urmăriți pas cu pas execuția acestui program pentru n=121 și m=4950.
2. Observați că nu am testat nicăieri dacă inițial n>m. Nu este necesar, deoarece, dacă inițial n ar fi mai mic decât m, după prima executare a instrucțiunii compuse subordonate instrucțiunii while, valorile variabilelor n și m vor fi interschimbate.

3. Modificați acest program astfel încât să afișeze etapele de determinare a c.m.m.d.c. ca în exemplul de mai sus.
4. Scrieți un program echivalent care să utilizeze instrucțiunea repetitivă `do while`.
5. Modificați acest program astfel încât să simplifice fracția `n/m` și să afișeze fracția ireductibilă obținută.
6. Modificați acest program astfel încât să verifice dacă numerele `n` și `m` sunt prime între ele.
7. Modificați acest program astfel încât să calculeze `c.m.m.m.c.(n,m)` (cel mai mic multiplu comun al numerelor `n` și `m`).
8. Modificați acest program astfel încât să determine $\Phi(n)$ (indicatorul lui Euler[14]). Prin definiție, $\Phi(n)$ este numărul de numere naturale mai mici decât `n`, relativ prime cu `n`. Două numere sunt relativ prime dacă cel mai mare divizor comun al lor este `1`.

Gard

Doi copii vopsesc un gard alcătuit din `n` scânduri (numerotate de la `1` la `n`, `n≤100000`) astfel: primul ia o cutie de vopsea roșie cu care vopsește scândurile cu numărul `p`, `2p`, `3p` etc. Al doilea procedează la fel, începe de la același capăt al gardului, dar ia o cutie de vopsea albastră și vopsește din `q` în `q` scânduri. Astfel, când vor termina de vopsit, gardul va avea multe scânduri nevopsite, unele scânduri vopsite în roșu, altele în albastru, iar altele în violet (cele care au fost vopsite și cu roșu și cu albastru).

Cunoscând numerele `n`, `p` și `q`, scrieți un program care să afișeze:

- câte scânduri rămân nevopsite;
- câte scânduri sunt vopsite în roșu;
- câte scânduri sunt vopsite în albastru;
- câte scânduri sunt vopsite în violet.

De exemplu, pentru `n=25`, `p=4`, `q=6` se vor afișa valorile `17 4 2 2`.

(Olimpiada Județeană de Informatică pentru Gimnaziu 2003, clasa a VI-a)

Soluție

O primă idee ar fi să parcurgem succesiv toate scândurile (adică toate numerele naturale de la `1` la `n`) și să verificăm pentru fiecare scândură următoarele cazuri:

- dacă numărul scândurii este divizibil cu `p` și este divizibil și cu `q`, atunci scândura va fi violet (și mărim cu `1` numărul de scânduri de culoare violet);
- dacă numărul scândurii este divizibil numai cu `p`, atunci scândura va fi roșie (și mărim cu `1` numărul de scânduri de culoare roșie);

14. Leonhard Euler (1707-1783) a fost unul dintre cei mai prolifici matematicieni din istorie. A scris 866 de lucrări și a câștigat de 12 ori Premiul Academiei Franceze.

- dacă numărul scândurii este divizibil numai cu q, atunci scândura va fi albastră (și mărim cu 1 numărul de scânduri de culoare albastră);
- dacă nici una dintre condițiile precedente nu a fost îndeplinită, atunci scândura va rămâne nevopsită (și mărim cu 1 numărul de scânduri nevopsite).

```
#include <iostream.h>
int main()
{long int n, i, p, q, nec, ros, alb, vio;
 cout<<"Numarul de scanduri n= "; cin>>n;
 cout<<"p, q = "; cin>>p>>q;
 nec=ros=alb=vio=0;
 for (i=1; i<=n; i++)
     if (i%p==0 && i%q==0) vio++;
        else
        if (i%p==0) ros++;
           else
           if (i%q==0) alb++;
              else nec++;
cout<<nec<<' '<<ros<<' '<<alb<<' '<<vio<<endl;
return 0;}
```

O idee mai eficientă ar fi să observăm că scândurile vopsite în violet (deci cele ale căror numere trebuie să fie divizibile cu p și cu q) trebuie, de fapt, să fie divizibile cu cel mai mic multiplu comun al numerelor p și q. Să notăm cu m c.m.m.m.c.(p,q). Observăm că numărul scândurilor violet este n/m. Numărul scândurilor roșii este n/p, din care eliminăm pe cele care au fost vopsite în violet, iar numărul scândurilor albastre este n/q, din care eliminăm pe cele care au fost vopsite în violet.

```
#include <iostream.h>
int main()
{long int n, m, p, q, a, b, r, nec, ros, alb, vio;
 cout<<"Numarul de scanduri n= "; cin>>n;
 cout<<"p, q = "; cin>>p>>q;
 a=p; b=q;          //calculam m=cmmmc(p,q)
 while (b) {r=a%b; a=b; b=r;}
 m=p*q/a;
 vio=n/m;
 ros=n/p - vio;
 alb=n/q - vio;
 nec=n-ros-alb-vio;
 cout<<nec<<' '<<ros<<' '<<alb<<' '<<vio<<endl;
 return 0;}
```

Exponent

Fie n un număr natural și p un număr prim. Să se determine cel mai mare număr natural k astfel încât p^k divide n.

Soluție

Problema cere practic să determinăm multiplicitatea factorului prim p în n.

```
#include <iostream.h>
int main ( void )
{int n, p, k=0;
 cout<<"n= "; cin>>n;
 cout<<"p= "; cin>>p;
 for (k=0; n%p==0; k++, n/=p);
 cout<<"k= "<<k<<endl;
 return 0; }
```

Termen Fibonacci

Fie șirul Fibonacci[15] 1, 1, 2, 3, 5, 8, 13, 21, 34,...

Observați că primii doi termeni sunt egali cu 1, fiecare termen următor fiind egal cu suma celor doi termeni care îl precedă.

Fie n un număr natural citit de la tastatură. Scrieți un program care afișează cel de-al n-lea termen din șirul Fibonacci.

Soluție

Observăm că pentru a calcula un termen al șirului Fibonacci avem nevoie de doi termeni precedenți. Vom utiliza trei variabile, f0, f1, și f2. În f0 și f1 vom reține doi termeni Fibonacci consecutivi, iar în f2 vom calcula termenul următor. Variabilele f0 și f1 vor fi inițializate cu 1 (primii doi termeni din șirul Fibonacci). Următorii termeni (de la al doilea până la al n-lea) îi vom calcula succesiv. La fiecare pas i, în f2 vom calcula al i-lea termen din șirul Fibonacci (f2=f1+f0), apoi ne pregătim pentru pasul următor (f0=f1, f1=f2):

```
#include <stdio.h>
int main(void)
{
int n, i, f0=1, f1=1, f2=1;
scanf("%d", &n);
for (i=2; i<=n; i++)
      { f2=f1+f0;    /* calculez termenul curent */
        f0=f1;       /* ma deplasez in sir       */
        f1=f2; }
printf("Cel de-al %d-lea termen Fibonacci este %d\n",n,f2);
return 0; }
```

15. Acest șir a fost introdus în 1202 de către Leonardus filius Bonaccii Pisanus (Leonardo, fiul lui Bonacci Pisanul) pentru rezolvarea următoarei probleme: Să se determine câte perechi de iepuri se vor naște într-un an dintr-o singură pereche de iepuri, știind că fiecare pereche de iepuri dă naștere lunar la o nouă pereche de iepuri, o pereche devine fertilă în decurs de o lună, iar iepurii nu mor pe parcursul anului. Șirul Fibonacci are multe aplicații, în variate domenii. De exemplu, numerele Fibonacci sunt utilizate în probleme de sortare și căutare sau în strategii de joc.

Exerciții

1. Urmăriți pas cu pas execuția acestui program pentru n=10.
2. Modificați algoritmul astfel încât să afișeze cel de-al n-lea termen din șirul Lucas[16]. Primii termeni din șirul Lucas sunt 1, 3, 4, 7, 11, 18, 29, 47,...
3. Modificați algoritmul astfel încât să calculeze suma primilor n termeni din șirul Fibonacci.
4. Modificați algoritmul astfel încât să calculeze cel mai mare termen din șirul Fibonacci mai mic sau egal cu n.

Existență

Se citesc de la tastatură două numere naturale nenule n și p, apoi se citesc succesiv n valori întregi. Scrieți un program care să verifice dacă printre cele n valori citite există multipli ai lui p.

Soluție

Vom citi succesiv cele n valori întregi în aceeași variabilă a. După fiecare citire vom verifica dacă valoarea citită este divizibilă cu p și, în caz afirmativ, vom reține faptul că există un multiplu al lui p. Rezultatul verificării îl vom reține în variabila exista. Inițial, valoarea acestei variabile este 0 (deoarece încă nu am descoperit nici un multiplu al lui p). Dacă pe parcursul citirii, vom descoperi un multiplu al lui p, variabila exista va primi valoarea 1.

```
#include <stdio.h>
int main(void)
{int n, p, a, exista=0, i;
 scanf("%d %d", &n, &p);
 for (i=1; i<=n; i++)
     {scanf("%d",&a); /* citesc un nou termen      */
      if (a%p==0)     /* am descoperit un multiplu */
         exista=1; }
 if (exista) printf("Exista un multiplu al lui %d", p);
    else printf("Nu exista un multiplu al lui %d", p);
 return 0; }
```

Exercițiu

1. Urmăriți pas cu pas execuția acestui program pentru șirul de valori 6, 3, 7, 4, 1, 12, 4, 5.
2. Modificați programul astfel încât să calculeze numărul de multipli ai lui p existenți în șir.

16. François-Edouard-Anatole Lucas (1842-1891) matematician francez cu importante contribuții în teoria numerelor.

Număr divizori primi

Se citește de la tastatură un număr natural nenul n, apoi se citesc succesiv n valori întregi. Scrieți un program care să verifice dacă oricare număr dintre cele n citite are un număr impar de divizori primi.

Soluție

Vom citi succesiv cele n valori în aceeași variabilă x. Pentru fiecare valoare citită, vom determina numărul de divizori primi (folosind un algoritm asemănător cu descompunerea în factori primi, numai că în loc să afișăm divizorii primi și multiplicitățile acestora, vom număra divizorii).

Rezultatul verificării îl vom reține în variabila oricare. Inițial, această variabilă are valoarea 1 (deoarece încă nu am găsit un număr care să nu aibă un număr impar de divizori primi). Dacă, pe parcursul citirii, vom întâlni un număr cu număr par de divizori primi, variabilei oricare îi vom atribui valoarea 0.

```
#include <stdio.h>
int main()
{unsigned n, x, d, nr, i, oricare=1;
 scanf("%u", &n);
 for (i=1; i<=n; i++)
    {scanf("%u",&x);       /* citesc o noua valoare      */
     nr=0;
     /*numarul de divizori primi ai lui x este initial 0 */
      d=2;            /*determin divizorii primi ai lui x */
      while (x>1)
          {if (x%d==0)        /* am descoperit un divizor */
              {nr++;
               /* numar acest divizor prim, apoi
               il elimin din x, prin impartiri repetate  */
               while (x%d==0) x/=d;  }
           d++; }             /* caut divizori mai departe */
     if (nr%2==0) oricare=0;} /* pentru ca am gasit o
                    valoare cu numar par de divizori primi */
if (oricare) printf("DA");
   else printf("NU");
return 0; }
```

Vocale

Se citește de la tastatură o propoziție scrisă cu litere mici, terminată cu '.'. Scrieți un program care calculează și afișează numărul de vocale din propoziție.

Soluție

Citim succesiv caracterele din care este constituită propoziția, până la sfârșit, în aceeași variabilă (să o numim c). Pentru fiecare caracter citit, verificăm dacă este vocală ('a', 'e', 'i', 'o' sau 'u') și dacă da, îl numărăm.

```
#include <stdio.h>
int main(void)
{unsigned nr=0;                          /* numarul de vocale */
 char c;/*in c vom citi succesiv caracterele propozitiei*/
 do
  {c=getchar();                          /*citim un caracter */
   if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
       nr++;}                          /* c este vocala, o numar */
 while (c!='.');
 printf("Numarul de vocale este %u\n",nr);
 return 0;}
```

Expresie

Fie n un număr natural şi x un număr real, citite de la tastatură. Să se calculeze valoarea expresiei[17]:

$$E(n,x)=1+\frac{x}{1!}+\frac{x^2}{2!}+...+\frac{x^n}{n!}$$

Soluție

Expresia este o sumă formată din termeni care au aceeaşi formă. Mai exact, termenul i din sumă ($i=\overline{0,n}$) este o fracție cu numitorul egal cu i! şi numărătorul egal cu x^i. Să urmărim calculul acestei expresii pentru n=3:

Pas	Factorial	Putere	Expresie
0	0!=1	x^0=1	1/1=1
1	1!=0!*1=1	x^1= x^0*x	1+x/1=1+x
2	2!=1!*2=2	x^2= x^1*x	1+x/1+x^2/2
3	3!=2!*3=6	x^3= x^2*x	1+x+x^2/2+x^3/6

Observăm că la fiecare pas i trebuie să calculăm i! (înmulțind factorialul de la pasul precedent cu i), x^i (înmulțind puterea de la pasul precedent cu x) şi să adăugăm un nou termen la expresie (x^i/i!).

```
#include <iostream.h>
int main()
{ unsigned long n, i, f;  float s, x, p;
  cout << "n, x= "; cin >> n >> x;
  for (f=s=p=i=1; i<=n; i++)
      { f*=i;          //actualizez factorialul
```

17. Această expresie este remarcabilă, deoarece poate fi utilizată pentru calculul aproximativ al lui ex (funcția exponențială).

```
        p*=x;          //actualizez puterea
        s+=p/f;    }   //adun termenul i la suma
    cout << "Expresia:" << s << endl;
    return 0; }
```

Radical

Fie x un număr real. Să se calculeze $\sqrt[3]{x}$ (radical de ordin 3 din x), folosind relația de recurență:

$r_1=1$

$r_i=(2*r_{i-1} + x/(r_{i-1}*r_{i-1}))/3$

Rezultatul trebuie să fie obținut cu o precizie mai bună decât 10^{-4}.

Soluție

Observăm că pentru a calcula termenul i din acest șir recurent este necesar numai termenul i-1. Prin urmare, vom utiliza două variabile r1 (în care reținem termenul calculat la pasul precedent) și r2 (în care reținem termenul pe care urmează să îl calculăm).

Pentru a obține precizia cerută în problemă, vom calcula termeni din șir până când diferența în valoare absolută dintre doi termeni consecutivi este mai mică decât precizia specificată. Pentru a calcula diferența în valoare absolută dintre doi termeni consecutivi, vom utiliza funcția fabs() din biblioteca de funcții matematice (această funcție calculează modulul unui număr de tip double specificat ca parametru).

```
#include <iostream.h>
#include <math.h>
#define EPS 0.0001
int main()
{ double x, r1, r2=1;
  cout<<"x= "; cin>>x;
  do
    {r1=r2;
     r2=(2*r1+x/(r1*r1))/3;}
  while (fabs (r2-r1)>=EPS);
  cout<< r2<<endl;
  return 0;}
```

Relație

Fie x un număr natural nenul. Să se determine toate perechile de numere naturale m, n care satisfac relația $m^2*(n+1)=x$.

Soluție

Observăm că pentru a fi satisfăcută această relație trebuie ca n+1 și m^2 să fie divizori ai lui x. Prin urmare, vom considera toate pătratele perfecte care divid pe x.

```
#include <iostream.h>
int main()
{int x, i;
 cout<<"x= "; cin>>x;
 for (i=1; i*i<=x; i++)
    if (x % (i*i)==0)
       cout<<"m= "<<i<<' '<<"n= "<<(x/(i*i)-1)<<endl;
 return 0; }
```

Sumă de două numere prime

Colegul de bancă a făcut la ora de matematică o afirmație interesantă: „orice număr natural n>3 se poate scrie ca sumă de două numere prime". El nu a demonstrat afirmația, așa că nu prea vă vine să credeți. Cum nu sunteți prea bun la matematică, v-ați gândit la o altă soluție: „inducția informatică". Prin urmare, scrieți un program care să citească VMax număr natural și care să verifice dacă pentru orice număr natural ≤Vmax este adevărată această proprietate.

Soluție

Parcurgem numerele naturale din intervalul [4,VMax] și pentru fiecare număr încercăm să determinăm o soluție. Pentru oricare număr n din intervalul specificat, considerăm toate perechile de forma (a, n-a), a≤n-a, până când găsim o pereche în care ambele valori sunt prime sau până când am epuizat toate perechile.

```
#include <iostream.h>
#include <math.h>
int main()
{int n, a, rad, d, prim, exista, VMax, oricare=1;
 cout <<"VMax= "; cin>>VMax;
 for (n=4; n<=VMax && oricare; n++)
     {for (exista=0, a=2; a<=n/2 && !exista; a++)
        {//testam daca a este prim
         prim=1;
         if (a>2 && a%2==0) prim=0;
            else
         for (rad=sqrt(a), d=3; prim&&d<=rad; d+=2)
             if (a%d==0) prim=0;
         if (prim)
            {//testam daca n-a este prim
             prim=1;
             if (n-a>2 && (n-a)%2==0) prim=0;
                else
                for (rad=sqrt(n-a),d=3; prim&&d<=rad; d+=2)
                    if ((n-a)%d==0) prim=0;
             if (prim) exista=1;} //am gasit o varianta
        }
      if (!exista) oricare=0; }
```

```
 if (oricare) cout<<"Proprietatea este verificata\n";
     else
     {cout<<"Proprietatea nu este verificata\n";
      cout<<"Pentru n= "<<n<<" nu exista solutie\n";}
 return 0;
}
```

Progres

Tatăl lui Gigel este foarte atent la performanțele școlare ale fiului său. El își notează toate notele obținute de fiul său, apoi realizează tot felul de statistici. Astăzi analizează șirul notelor și vrea să determine lungimea celei mai lungi secvențe crescătoare, secvență de note care corespunde unei perioade de progres școlar. Cum notele sunt deja foarte multe, el are nevoie de un program care să citească numărul de note și secvența notelor și să afișeze lungimea celei mai lungi secvențe crescătoare de note.

Soluție

Vom citi succesiv notele. Pentru a putea compara două note consecutive, vom utiliza două variabile: în variabila `a` reține nota precedentă, iar în variabila `b` reține nota curentă. La fiecare nouă citire, compar nota curentă cu cea precedentă. Dacă nota curentă este cel puțin egală cu nota precedentă, secvența curentă de note este crescătoare, deci măresc lungimea ei (memorată în variabila `lg`). Dacă nota curentă este mai mică decât nota precedentă, deducem că s-a terminat o secvență crescătoare, compar lungimea ei cu lungimea maximă (memorată în variabila `lgmax`) și reinițializez variabila `lg` cu `1` (nota curentă este prima notă dintr-o altă secvență).

```
#include <iostream.h>
int main()
{int n, i, a=0, b, lg=0, lgmax=0;
 cout<<"nr de note = "; cin>>n;
 cout<<"Introduceti notele: ";
 for (i=1; i<=n; i++)
     {cin>>b; //citesc o nota
      if (a<=b) lg++; //maresc lungimea secventei curente
          else        //s-a terminat o secventa crescatoare
           {if (lgmax<lg) //compar lg cu lungimea maxima
               lgmax=lg;
            lg=1;}        //incepe o noua secventa
     a=b; //nota curenta devine nota precedenta
    }
 //compar si lungimea ultimei secvente
 if (lgmax<lg) lgmax=lg;
 cout<<"Lungimea maxima = "<<lgmax<<endl;
 return 0;}
```

Tabla adunării

Scrieți un program care să îl ajute pe un copil de clasa I să învețe tabla adunării până la 100. Programul va genera aleator două numere naturale mai mici decât 100, le va afișa pe ecran și îl va întreba pe copil cu cât este egală suma lor. Dacă răspunsul copilului este corect, îl felicită. Dacă răspunsul copilului este greșit, îl întreabă dacă dorește să mai încerce sau dacă dorește să afle răspunsul corect. Programul repetă testarea până copilul se plictisește și spune că nu mai vrea!

Soluție

Programul va genera aleator două numere întregi mai mici decât 100. Pentru aceasta, va trebuie să inițializăm generatorul de numere aleatoare, cu ajutorul funcției `randomize()` declarate în fișierul antet `stdlib.h`:

```
void randomize(void);
```

ceea ce înseamnă că funcția nu are parametri și nu returnează nici un rezultat.

Pentru a genera un număr aleator, vom utiliza funcția `random()`, declarată în fișierul antet `stdlib.h`:

```
int random(int x);
```

Funcția returnează o valoare aleatoare de tip `int`, cuprinsă între `0` și `x-1`.

După generarea aleatoare a celor două numere și afișarea lor pe ecran, programul va citi răspunsul copilului și îl va compara cu suma celor două numere:

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
int main ()
{ char raspuns;
  int x, y, suma;
  cout<<"Sa invatam tabla adunarii! Esti pregatit? (D/N)";
  raspuns=getche();
  if (raspuns =='N' || raspuns =='n')
      cout<<"\nMai gandeste-te! Eu zic sa incercam!\n";
  randomize();//initializez generatorul de numere aleatoare
  do
    {x=random(100);y=random(100);//generez aleator numerele
     cout<<'\n'<<x<<'+'<<y<<"=?\b"; raspuns = 'D';
     do
       {cin >> suma;     //citeste raspunsul elevului
        if (suma==x+y) cout << "\n Excelent raspuns!!!";
           else
           {cout <<"\nNu! Mai incerci o data?(D/N) ";
            raspuns=getche();}
       }
     while (suma!=x+y && (raspuns=='D'|| raspuns =='d'));
     if (raspuns=='N'|| raspuns =='n')
```

```
        cout <<"\nRaspunsul corect era "<<x+y<<endl;
      cout <<"\nContinuam? (D/N) "; raspuns=getche();
    }
  while (raspuns=='D'||raspuns =='d');
  cout <<"La revedere!!!"<<endl;
  return 0;}
```

Exercițiu

Utilizând funcții din fișierul antet `conio.h`, faceți acest program cât mai atrăgător (scrieți mesajele colorat pe o culoare de fond vioaie, realizați și puțină animație). Dacă doriți, puteți utiliza și mici semnale sonore (cu ajutorul funcțiilor `sound()` și `nosound()` din fișierul antet `dos.h`).

Semne

Fie `n` un număr natural nenul citit de la tastatură. Scrieți un program care să determine un număr natural `k` și o combinație de $\overline{1,k}$ semne + sau - (mai exact o succesiune $x_1, x_2, ..., x_k$ unde $x_i \in \{-1, 1\}$, $\forall i=\overline{1,k}$) astfel încât să aibă loc relația: $n=x_1*1^2+x_2*2^2+...+x_k*k^2$. Programul va afișa pe ecran o succesiune de `k` semne + sau - care să îndeplinească relația de mai sus. De exemplu, pentru `n=3`, o soluție este `-+--+`. Pentru `n=2`, o soluție este `---+`.

Soluție

Analizând problema pentru câteva cazuri particulare, observăm că succesiunea de semne `+--+` produce întotdeauna valoarea `4`, deoarece:

$k^2-(k+1)^2-(k+2)^2+(k+3)^2=4, \ \forall k>0.$

Prin urmare, orice număr `n` divizibil cu `4` se poate scrie ca o succesiune de `n/4` secvențe `+--+`. Dacă numărul nu este divizibil cu `4`, vom scrie în prealabil o secvență de semne egală cu valoarea restului.

```
#include <iostream.h>
int main()
{ int n;
  cout << "n= "; cin >> n;
  switch (n%4)
  { case 1: cout << "+"; break;
    case 2: cout << "---+"; break;
    case 3: cout << "-+--+";  }
  for (int i=0; i<n/4; i++) cout << "+--+";
  return 0; }
```

Excursie

La Olimpiada Națională de Informatică, organizatorii vor să realizeze o excursie în care să participe toți cei `n` concurenți. Fiecare concurent are asociat în mod unic

un număr între 1 și n. Sponsorii au pus la dispoziție trei mijloace de transport impunând însă o condiție: *suma numerelor de ordine ale participanților din fiecare mijloc de transport să fie aceeași*, în caz contrar, excursia nu va avea loc.

Scrieți un program care să distribuie pe cei n concurenți în cele trei mijloace de transport.

(Concursul Interjudețean „Urmașii lui Moisil", Iași, 2001)

Soluție

În primul rând, observăm că suma valorilor din fiecare mașină trebuie să fie `n*(n+1)/6` (pentru că `1+2+3+...+n=n*(n+1)/2`).

În al doilea rând, observăm că există cazuri pentru care nu există soluție (pentru `n<5` sau pentru acele valori ale lui n pentru care suma `n*(n+1)/2` nu este divizibilă cu 3, pentru că trebuie să obținem 3 grupe de sumă egală).

Analizăm separat cazurile `n=5, 6, 8, 9` și observăm că pentru fiecare dintre acestea există soluție:

n	n=5	n=6	n=8	n=9
Grupe	(5) (2,3) (1,4)	(2,5) (3,4) (1,6)	(1,2,3,6) (5,7) (4,8)	(1,2,3,4,5) (7,8) (6,9)

Pentru orice altă valoare a lui n pentru care există soluție, construim soluția plecând de la unul dintre aceste 4 cazuri elementare, adăugând grupe de câte 6 elemente consecutive, distribuite astfel: (a+1, a+6), (a+2, a+5), (a+3, a+4).

```
#include <iostream.h>
int main()
{long int n, Start, x, i;
 cin>>n;
 if (n<5||n*(n+1)%3) {cout<<"Nu exista solutie"; return 0;}
 if (!((n-9)%6)){Start=10; x=(n-9)/6;}
 if (!((n-8)%6)){Start= 9; x=(n-8)/6;}
 if (!((n-6)%6)){Start= 7; x=(n-6)/6;}
 if (!((n-5)%6)){Start= 6; x=(n-5)/6;}
 switch (Start)                                          //masina 1
  {case 10: cout<<"1 2 3 4 5 "; break;
   case  9: cout<<"1 2 3 6 "; break;
   case  7: cout<<"3 4 "; break;
   case  6: cout<<5<<' '; break;                    }
 for (i=0;i<x;i++) cout<<Start+6*i<<' '<<Start+5+6*i<<' ';
 cout<<endl;
 switch (Start)                                          //masina 2
   {case 10: cout<<7<<' '<<8<<' '; break;
    case  9: cout<<5<<' '<<7<<' '; break;
    case  7: cout<<2<<' '<<5<<' '; break;
    case  6: cout<<2<<' '<<3<<' '; break;          }
for (i=0;i<x;i++) cout<<Start+1+6*i<<' '<<Start+4+6*i<<' ';
cout<<endl;
```

```
switch (Start)                                          //masina 3
 {case 10: cout<<6<<' '<<9<<' '; break;
  case  9: cout<<4<<' '<<8<<' '; break;
  case  7: cout<<1<<' '<<6<<' '; break;
  case  6: cout<<1<<' '<<4<<' '; break;          }
for (i=0;i<x;i++) cout<<Start+2+6*i<<' '<<Start+3+6*i<<' ';
cout<<endl;
return 0; }
```

2.10. *Probleme propuse*

1. Fie x un număr real. Scrieți un program C care să calculeze valoarea funcției: $F(x)=\max\{|2x-1|, 9-x^2\}$.
2. Care este efectul următoarei secvențe de instrucțiuni?

```
int a, b, c, x;
a=3; b=5; c=7;
if (a-b/2<0) x=1; else if (a+b-c/2<b) x=2; else if
(a%b+c>b) x=3; else x<-4;
cout<<x;
```

3. Ce valoare inițială ar putea avea variabila x, astfel încât la sfârșitul execuției următoarei secvențe de instrucțiuni variabila y să aibă valoarea 2?

```
if (x>3) if (x<7) if (x%2==0) y=1; else y=2; else y=3;
else y=4;
```

4. Fie a și b două numere întregi. Scrieți un program care să verifice dacă a și b sunt numere consecutive.
5. Fie ecuația cu coeficienți reali $ax^2+bx+c=0$ ($a\neq 0$). Scrieți un program care, fără a calcula rădăcinile ecuației, să determine natura și semnul acestora.
6. Fie x un număr natural de trei cifre. Scrieți un program care să elimine una dintre cifrele numărului astfel încât numărul de două cifre rămas să fie maxim.
7. Scrieți un program care să citească 3 caractere și să determine câte caractere distincte s-au citit.
8. Fie a și b două unghiuri, ale căror măsuri sunt exprimate în grade, minute și secunde. Să se scrie un program care să calculeze și să afișeze măsura sumei celor două unghiuri.
9. Se citesc de la tastatură a, b și c, trei numere reale pozitive. Scrieți un program care să verifice dacă numerele citite pot constitui laturile unui triunghi dreptunghic. În caz afirmativ, calculați și afișați aria triunghiului.
10. Fie x și y două numere reale, citite de la tastatură. Scrieți un program care calculează și afișează valoarea funcției:

$$f(x,y)=\begin{cases}\dfrac{x+y}{5xy}, \text{dacă } x,y>0\\ \min(x,y), \text{dacă } x=0 \text{ sau } y=0\\ \left(\dfrac{1}{x}+\dfrac{1}{y}\right)\left(\dfrac{1}{x}+\dfrac{1}{y}+x^2+y^2\right), \text{altfel}\end{cases}$$

11. Scrieți un program care citește de la tastatură trei numere întregi strict pozitive `a`, `b` și `c`, numere cu cel mult trei cifre fiecare. Valoarea variabilei `a` reprezintă distanța în km dintre orașul A și orașul B, `b` distanța în km dintre orașul B și orașul C, iar `c` reprezintă distanța în km dintre orașul C și orașul A. Știind că un călător își planifică o vizită a celor trei orașe pornind din oricare din orașele A sau B și ajungând, în final, în oricare din orașele B sau C cu trecere prin cel de-al treilea oraș, să se determine un traseu de lungime minimă care respectă aceste condiții. Programul va afișa cele trei litere corespunzătoare celor trei orașe, în ordinea în care sunt vizitate. Se va alege o metodă cât mai eficientă din punctul de vedere al gestionării memoriei. De exemplu, pentru `a=58`, `b=140`, `c=125`, se va afișa `BAC`.

(Bacalaureat 2002, sesiune specială)

12. Să se scrie un program care să rezolve sistemul de două ecuații de gradul I, cu două necunoscute și coeficienți reali:

$$\begin{cases}a_1x+b_1y=c_1\\ a_2x+b_2y=c_2\end{cases}$$

13. Un elev este declarat promovat la Bacalaureat dacă la fiecare dintre cele cinci probe de examen a luat cel puțin nota 5, iar media sa generală este cel puțin 6. Date fiind cele 5 note pe care elevul le-a obținut la bacalaureat, scrieți un program care să verifice dacă elevul a promovat sau nu examenul de Bacalaureat.

14. Ionel are `H1` cm, Gigel are `H2` cm, iar Dănuț are `H3` cm. Scrieți un program care să afișeze numele celor trei copii în ordinea crescătoare a înălțimii.

15. Care este efectul următoarelor secvențe de instrucțiuni?

a.

```
int x=0, a=6, b=4, c=5, d=1;
if (a>b) if (b>c) if (c>d) x=1; else x=2;
      else x=3; else if (a>c) x=4; else x=5;
cout  << x;
```

b.

```
int n=5;
while (--n) cout << n;
```

c.

```
int n=0;
while (--n) cout << n;
```

d.
```
int n=20;
for (int i=0; i<n; i++)
    if (!(i%5)) cout<<i;
```

e.
```
for (int i=0; i<5; i++)
    for (int j=i+1; j<5; j++)
        cout<<i<<j<<' ';
```

f.
```
int n=2;
do if (n%2) cout << n++;
while (++n % 2);
```

g.
```
int n=-5;
do if (n++) break; while (1);
cout << n;
```

h.
```
int n=8, i=0, j=1;
for (int k=0; k++ < n; j=i+j, i=j-i)
    cout << j << ' ';
```

16. Care este valoarea variabilei întregi `a` pentru care următoarea secvență de program va afișa exact un caracter `'A'`?

(variantă Bacalaureat, 2000)

```
x=5;
do   {cout << 'A'; x++; } while (x>a);
```

a. 12 **b.** 5 **c.** 6 **d.** 4 **e.** 1

17. Fie `a` și `b` două numere naturale, $a \leq b$. Care dintre următoarele secvențe de instrucțiuni calculează în variabila `nr` numărul de valori pare din intervalul `[a,b]`?

a.
```
nr=0;
for (i=a; i<=b; i++)
    if (!i%2) nr++;
```

b.
```
nr=0;
while (b>=a)
   if (b%2==0) nr++; b--;
```

c.
```
nr=(b-a+1)/2+(b-a+1)%2;
```

d.
```
nr=(b-a+1)/2;
if (a%2 && b%2) nr--;
```

e.
```
nr=(b-a+1)/2;
if (a%2==0&&b%2==0) nr++;
```

f.
```
for (nr=i=0; i<=b-a;i++)
      if ((i+a)%2==0) nr++;
```

18. Se consideră următoarea secvență de instrucțiuni:

```
while(i%2)
     {cout<<i%2;
     i/=2;}
```

Care dintre valorile următoare pot fi considerate valoare inițială pentru variabila `i`, astfel încât pe ecran să se afișeze `111`?

a. 7 **b.** 128 **c.** 15 **d.** 135 **e.** 117 **f.** 39

19. Câte caractere # afișează secvența următoare?

```
int a, b, c, n=5;
for (a=1; a<=n; a++)
     for (b=1; b<=n; b+=2)
          for (c=b+1; c<=n; c++)
               cout<<'#';
```

20. Care dintre următoarele secvențe de instrucțiuni atribuie variabilei de tip `int x` valoarea 2^n, cu `n` număr natural (`n<16`), variabila auxiliară `i` fiind de tip `int`?

a.
```
x=1;
for (i=1; i<=n; i++) x *= n;
```

b.
```
x=1;
for (i=0; i<n; i++) x *= 2;
```

c.
```
x=1;
for (i=1; i<=n; i++) x <<= 1;
```

d.
```
x=2;
for (i=1; i<=n; i++) x*=i;
```

e.
```
x=1; i=0;
while (i<n) x*=2; i++;
```

f.
```
x=1;
x << n;
```

21. Ce valoare inițială ar trebui să aibă variabila întreagă `x` astfel încât după execuția următoarei secvențe pe ecran să fie afișată valoarea `2`?

```
y=0;
while (x)
 {y+=x%10; x=x/10;}
if (y<5) if (y>2) if (y>4) cout<<1; else cout<<2; else
cout<<3; else cout<<4;
```

a. 111 **b.** 21 **c.** 113 **d.** 3 **e.** 2 **f.** 312

22. Fie `n` și `i` două variabile întregi. Ce se va afișa pe ecran după executarea următoarei secvențe de instrucțiuni?

```
for (n=9, i=0; i<=n; i++) {i++; n--;}
cout<<n<<i;
```

a. 910 **b.** 99 **c.** 58 **d.** 67 **e.** 57 **f.** 56

23. Secvența următoare trebuie să testeze dacă numărul natural `x` este pătrat perfect.

```
for (ok=1, d=2; x>1 && ok; )
    {for (m=0; x%d==0; m++, x/=d);
     if  (...) ok=0; else d++;     }
if (ok) cout<<"Este patrat perfect.\n";
   else cout<<"Nu este patrat perfect.\n";
```

Cum trebuie completată secvența care lipsește (marcată cu . . .)?

a. `m==0` **b.** `m==1` **c.** `m%2==0` **d.** `m%2`

24. Completați secvențele lipsă (marcate cu ...[1] și ...[2]) din algoritmul următor, astfel încât să se afișeze numărul de numere naturale din intervalul [x,y] care sunt prime cu y.

```
int x, y, z, i, a, b;
cin>>x>>y;
for (z=0, i=x; i<=y; i++)
 {a=i; b=y;
 while (...[1])
 if (a>b) a=a-b; else b=b-a;
 if (...[2]) z++; }
cout << z;
```

a.	b.	c.	d.
1. a!=y	1. a>y	1. a!=b	1. b==0
2. a==1	2. a==y	2. a==1	2. a!=1

25. Știind că valoarea inițială a variabilei întregi `i` este mai mare decât 10, stabiliți care este valoarea expresiei `abs(3-i)` la sfârșitul executării următoarei instrucțiuni.

(variantă Bacalaureat, 2002)

```
while (i>4) i--;
```

a. -1 **b.** 0 **c.** o valoare nedeterminată
d. 1 **e.** 2 **f.** o valoare mai mare decât 2.

26. Se consideră următoarea secvență de instrucțiuni:

```
int a, b, c=0;
cin>>a>>b;
while (a && b)
      {if (a%10>b%10) c=c*10+a%10;
          else  c=c*10+b%10;
       a/=10;  b/=10;                }
```

a. Ce valoare va avea variabila `c` după execuția acestei secvențe dacă valorile citite pentru variabilele `a` și `b` sunt `14624` și, respectiv, `8632` ?

b. Dacă se citește valoarea `7382` pentru variabila `a`, ce valoare am putea introduce pentru variabila `b` astfel încât valoarea calculată în variabila `c` să fie `2938`?

c. Dați exemplu de două numere naturale de cel puțin două cifre, care ar putea fi introduse pentru variabilele `a` și `b`, astfel încât valoarea calculată în variabila `c` să fie `1`?

27. Se consideră următoarea secvență de instrucțiuni:

```
int n, d=2;
cin>>n;
while (n>1)
      if (n%d==0) n/=d;
         else d++;
cout<<d;
```

a. Ce valoare va fi afișată dacă valoarea citită pentru variabila n este 720?
b. Dați exemplu de valori (cel puțin două) care ar putea fi introduse pentru variabila n, astfel încât valoarea afișată să fie 11.
c. Care este efectul acestei secvențe de instrucțiuni?

28. Se consideră următoarea secvență de instrucțiuni:

```
int n, b, x=0, p=1;
cin>>n>>b;                    /* 2≤b≤9 */
while (n)
      x+=p*(n %10), p*=b, n/=10;
cout<<x;
```

a. Ce valoare va fi afișată dacă se citesc valorile 1032 și 4?
b. Ce valoare trebuie să introducem pentru variabila b astfel încât pentru n=407 să se afișeze valoarea 331?
c. Care este efectul acestei secvențe de instrucțiuni?

29. Se consideră următoarea secvență de instrucțiuni:

```
int x, a, b, i;
cin>>a>>b;
i=a; x=0;
while (i<=b) { x+=i;i++; }
cout<<x;
```

a. Ce valoare va fi afișată pe ecran dacă se introduc valorile 1 și 10?
b. Ce valori ar putea fi introduse pentru a și b astfel încât programul să afișeze valoarea 22?
c. Dați exemplu de valori distincte care ar putea fi introduse astfel încât programul să afișeze valoarea 0.
d. Scrieți un program echivalent mai eficient.

30. Care dintre următoarele secvențe de instrucțiuni atribuie variabilei întregi u valoarea primei cifre a numărului natural reprezentat de variabila x ?

a.
```
u=x/10;
```
c.
```
u=x%10;
```
b.
```
u=x; while (u>=10) u=u%10;
```
d.
```
while (x>=10) x=x/10; u=x;
```

31. Care dintre următoarele secvențe afișează cel mai mare divizor propriu al numărului natural neprim memorat în variabila x?

a.
```
d=2;
while (x%d==0) d++;
cout<<x/d;
```
b.
```
d=2;
do d++; while (x%d);
cout<<x/d;
```
c.
```
d=2;
while (x%d) d++;
cout<<x/d;
```
d.
```
d=x/2;
while (x%d) d--;
cout<<d;
```

32. Ce valoare inițială ar trebui să aibă variabila x astfel încât după execuția următoarei secvențe de instrucțiuni să se afișeze valoarea 640?

```
d=2;
while (d<50) { x=x*d; d=d*d;}
cout<<x;
```

33. Fie n un număr natural nenul și p un număr prim. Următoarea secvență de instrucțiuni trebuie să determine multiplicitatea lui p în n. Ce greșeli conține această secvență?

```
int n, p, m;
cin>>n>>p;
while (n%p==0) m++; n/=p;
cout<<m;
```

34. *Concurs*

La un concurs de matematică se acordă câte 5 puncte pentru fiecare problemă rezolvată corect și se scad 3 puncte pentru fiecare problemă rezolvată greșit. Un elev a obținut x puncte. Scrieți un program care să determine câte probleme a rezolvat corect și câte probleme a greșit elevul. Dacă există mai multe soluții, o veți afișa pe cea cu număr minim de probleme greșite.

35. *Numere prietene*

Două numere naturale a și b se numesc prietene dacă a este egal cu suma divizorilor lui b (exclusiv b), iar b este egal cu suma divizorilor lui a (exclusiv a). De exemplu, a=220 și b=284 sunt prietene. Scrieți un program care să determine primele trei perechi de numere prietene, cu a<b.

36. *Expresii*

Fie n un număr natural și x un număr real, citite de la tastatură. Să se calculeze valoarea expresiei[18]:

a. $E_1(n,x)=\frac{x}{1!}-\frac{x^3}{3!}+\frac{x^5}{5!}-\ldots+(-1)^n\frac{x^{2n+1}}{(2n+1)!}$

b. $E_2(n,x)=1-\frac{x^2}{2!}-\frac{x^4}{4!}-\ldots+(-1)^n\frac{x^{2n}}{(2n)!}$

37. *Prime*

Fie x un număr natural, x>2. Scrieți un program care să determine cel mai mare număr prim mai mic decât x și cel mai mic număr prim mai mare decât x.

18. Aceste expresii sunt de asemenea remarcabile, deoarece (așa cum veți demonstra la analiză matematică) pot fi utilizate pentru calculul aproximativ al lui sin(x), respectiv cos(x).

38. *Sumă și produs*

Se citește de la tastatură un număr natural nenul n, apoi se citesc succesiv n valori întregi. Scrieți un program care determină și afișează suma valorilor pare citite și produsul valorilor nenule.

39. *Palindrom prim*

Se citește de la tastatură un număr natural nenul n, apoi se citesc succesiv n valori naturale. Să se verifice dacă printre valorile citite există un palindrom prim.

40. *Cifre nenule*

Se citește un număr natural nenul n, apoi se citesc succesiv n valori întregi. Să se verifice dacă oricare dintre cele n valori citite are exact 3 cifre nenule.

41. *Cifra 0*

Scrieți un program care să citească de la tastatură o succesiune de valori naturale, până la citirea valorii –1, și care să determine de câte ori apare cifra 0 în scrierea numerelor citite.

42. *Cifra de control*

Fie n un număr natural, citit de la tastatură. Scrieți un program care calculează și afișează cifra de control a lui n. Cifra de control a unui număr natural se obține calculând suma cifrelor numărului, apoi suma cifrelor sumei, ș.a.m.d. până la obținerea unei singure cifre.

De exemplu, pentru n=293 calculăm suma cifrelor 2+9+3=14. Cum suma nu este formată dintr-o singură cifră, repetăm procedeul: 1+4=5. Deci 5 este cifra de control a lui 293.

43. *Cifre*

Fie N un număr natural nenul (N≤1000000000). Scriind în ordine, unul după altul, toate numerele naturale de la 1 la N, obținem o secvență de cifre. De exemplu, pentru N=22, obținem:

12345678910111213141516171819202122

Scrieți un program eficient care să determine numărul de cifre din secvența formată din numerele de la 1 la N.

44. *Pătrate perfecte*

Se citește de la tastatură un număr natural n. Scrieți un program eficient care să afișeze toate pătratele perfecte mai mici decât n.

45. *Numere piramidale*

Numerele piramidale se definesc ca fiind sumele parțiale ale șirului pătratelor perfecte 1, 4, 9, 16, 25... De exemplu, primele 5 numere piramidale sunt 1, 5, 14, 30, 55. Scrieți un program care să afișeze primele n numere piramidale.

46. *Fracții*

Fie n un număr natural nenul. Scrieți un program care să determine numărul de fracții ireductibile care au numitorul și numărătorul din mulțimea {1, 2, ..., n}. De exemplu, pentru n=3, se pot forma fracțiile ireductibile { $\frac{1}{1}$, $\frac{1}{2}$, $\frac{1}{3}$, $\frac{2}{1}$, $\frac{2}{3}$, $\frac{3}{1}$, $\frac{3}{2}$ }, deci programul va afișa valoarea 7.

47. *Șir*

Să considerăm următorul șir definit prin recurență astfel $s_0=1$, $s_1=2$, $s_2=1$, iar $s_n=s_{n-1}+2*s_{n-2}-s_{n-3}$ (pentru $n\geq 3$). Scrieți un program care să afișeze al n-lea termen din acest șir.

48. *Medie semestrială*

Se citește n, numărul de note obținute de un elev la informatică pe parcursul semestrului, apoi se citesc cele n note. Se citește apoi nota obținută de elev la teză. Scrieți un program care să calculeze media semestrială a elevului.

49. *C.m.m.d.c.*

Se citește n, un număr natural nenul, apoi se citesc n valori naturale. Scrieți un program care să calculeze cel mai mare divizor comun al celor n numere citite.

50. *Număr*

Se citește de la tastatură o succesiune de caractere, terminată cu spațiu (' '). Scrieți un program care să verifice dacă succesiunea de caractere citită poate fi considerată un număr natural scris în baza 10. Modificați programul astfel încât să testeze dacă succesiunea de caractere citită poate fi considerată un număr real.

51. *Cuvinte*

Se citește de la tastatură o propoziție constituită din cuvinte separate prin unul sau mai multe spații. Sfârșitul propoziției este marcat de întâlnirea caracterului '.'. Scrieți un program care determină și afișează numărul de cuvinte din propoziție.

52. *Progresie aritmetică*

Se citesc două numere naturale x și r, care reprezintă primul termen, respectiv rația unei progresii aritmetice, apoi se citește un număr natural n. Scrieți un program care să afișeze primii n termeni ai acestei progresii aritmetice.

Fie n un număr natural nenul. Scrieți un program care să determine cel mai mic număr natural care se împarte exact la toate numerele naturale mai mici sau egale cu n.

53. *Număr maxim de divizori primi*

Se citește de la tastatură un număr natural n, apoi se citesc succesiv n numere naturale. Să se determine un număr dintre cele n citite, care are un număr maxim de divizori primi.

54. *Palindroame prime*

Să se scrie un program care să citească un număr natural n ($n \leq 1000000000$) și care să determine eficient toate palindroamele prime mai mici decât n.

55. *Sumă și produs*

Fie n un număr natural. Să se determine toate numerele naturale mai mici decât n cu proprietatea că sunt divizibile atât prin suma, cât și prin produsul cifrelor lor.

56. *Zerouri*

Se citesc de la tastatură n ($n \leq 1000000$) numere naturale (<30000). Să se determine numărul de zerouri în care se termină produsul numerelor citite.

57. *Problema bibliotecarului*

Bibliotecarul de la școala ta a descoperit o carte în care paginile ce ar fi trebuit să fie numerotate cu un număr prim aveau numărul de pagină mâzgălit. Știind că numerotarea paginilor cărții începe de la 3, scrieți un program care citește de la tastatură pe n numărul de pagini din carte și afișează pe ecran numărul de pagini nemâzgălite, precum și numărul de cifre pe care trebuie să le scrie bibliotecarul, pentru a renumerota paginile mâzgălite.

58. *Numere bine ordonate*

Se numesc numere „bine ordonate" acele numere care au cifrele în ordine strict crescătoare sau strict descrescătoare. Să se scrie un program care determină toate numerele „bine ordonate crescător" cu trei cifre. Modificați programul pentru a obține și toate numerele „bine ordonate descrescător" cu trei cifre. Câte soluții există?

59. *Relație*

Fie n un număr natural. Să se afle numerele naturale de trei cifre xyz care au proprietatea că $n/(x^2+y^2+z^2)$ este număr natural.

60. *Cifre romane*

Se citește de la tastatură un număr natural $n<5000$. Scrieți un program care afișează numărul n în scriere cu cifre romane.

Indicație: I=1; V=5; X=10; L=50; C=100; D=500; M=1000; IV=4; IX=9; XL=40; XC=90; CD=400; CM=900.

61. *Plata sumei*

Se citesc de la tastatură două numere naturale S și x ($0<S\leq 10000$, $0<x\leq 100$). S reprezintă o sumă pe care trebuie să o plătim, utilizând un număr minim de bancnote. Bancnotele au ca valori numai puteri ale lui x ($1, x, x^2, x^3, \ldots$) și presupunem că dispunem de un număr suficient de mare de bancnote. Scrieți un program care afișează pe ecran o modalitate de a plăti suma, utilizând un număr minim de bancnote. De exemplu, pentru S=107 și x=5, veți afișa:

```
4 bancnote cu valoarea 25
1 bancnote cu valoarea 5
2 bancnote cu valoarea 1
```

62. *Euro-dolar*

Ion are 100 de euro şi un prieten care l-a ajutat să obţină ratele de schimb euro-dolar pentru următoarele n zile (n≤100). Mai exact, pentru fiecare dintre cele n zile, prietenul îi comunică două numere D şi E (ceea ce înseamnă că, în ziua respectivă, cu 100 de euro se pot cumpăra D dolari, iar cu E dolari se pot cumpăra 100 euro). Scrieţi un program care să determine suma maximă (exprimată în euro) pe care Ion o poate acumula după n zile, făcând tranzacţii euro-dolar.

De exemplu, să considerăm că n=8 şi ratele de schimb pentru cele 8 zile sunt:

```
218 219
228 231
227 235
205 213
230 232
239 239
251 258
205 213
```

Pentru a acumula o sumă maximă, Ion va realiza următoarele tranzacţii:

```
Ziua 2 ... schimba 100.0000 Euro in 228.0000 USD
Ziua 4 ... schimba 228.0000 USD in 107.0422 Euro
Ziua 7 ... schimba 107.0422 Euro in 268.6760 USD
Ziua 8 ... schimba 268.6760 USD in 126.1389 Euro
```

63. *Şef*

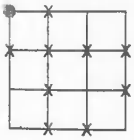
O firmă face angajări. Fiecărei persoane angajate îi corespunde un număr natural nenul a cărui reprezentare binară ocupă 16 biţi. Acest număr reprezintă codul persoanei în ierarhia firmei. Astfel, fiind date două numere a şi b, spunem că a este „şeful" lui b dacă pentru orice poziţie binară 0 a lui a, poziţia corespunzătoare din b este 0. Evident, orice persoană este propriul ei şef. Se citeşte n, un număr natural care reprezintă codul unei persoane. Se cere să se afişeze numărul şefilor săi. De exemplu, pentru n=255 veţi afişa 256.

(Olimpiada Judeţeană de Informatică, Timiş, 2000)

64. *Ceas cu cuc*

Bunica mea are un ceas cu cuc care cântă la ore fixe (de un număr de ori egal cu ora) şi la „şi jumătate" (o singură dată). Eu am plecat de acasă exact după ce cucul a cântat de oră fixă. Când m-am întors, cucul tocmai cântase de „şi jumătate" şi bunica supărată mi-a spus: „De când ai plecat, cucul a cântat de x ori!". Fiindcă nu înţeleg de ce bunica mea e aşa de supărată, scrieţi un program care să determine cât timp am lipsit de acasă!

65. *Problema pădurarului neamț*



Să ne imaginăm o pădure nemțească ca pe o rețea ortogonală[19] de `n x n` puncte (`n≤100`), astfel încât distanța dintre oricare două puncte vecine pe orizontală sau verticală este egală cu `1`. În toate punctele se află pomi, cu excepția unui singur punct de coordonate `x0`, `y0` în care se află pădurarul. Presupunând că pomii sunt punctiformi, pădurarul vede doar o parte din pomi (pe orice linie dreaptă care trece prin poziția sa pădurarul vede cel mult `2` pomi). Scrieți un program care citește de la tastatură `n`, `x0` și `y0` și afișează numărul de pomi pe care îi vede pădurarul. De exemplu, pentru `n=4`, `x0=0`, `y0=0`, pădurarul vede `9` pomi.

66. *Numai cu 0 și 1*

Fie `n` un număr natural (`n≤30000`) și `b` o bază (`2≤b≤10`). Să se scrie un program care afișează în baza `10` toate numerele mai mici sau egale cu `n`, care scrise în baza `b` folosesc numai cifrele `0` și `1`. De exemplu, pentru `n=64` și `b=4`, programul va afișa `0, 1, 4, 5, 16, 17, 20, 21, 64`.

19. Vă puteți imagina o rețea ortogonală ca pe o foaie de matematică. Punctele sunt la intersecțiile liniilor orizontale cu cele verticale.

3. Fişiere

3.1. *Noţiuni introductive*

Un *fişier* este o colecţie de date de acelaşi tip, memorate pe suport extern (hard-disc, dischetă, CD etc).

Avantajele utilizării fişierelor sunt o consecinţă a proprietăţilor memoriilor externe. Fişierele permit memorarea unui volum mare de date persistente (care nu se „pierd" la încheierea execuţiei programului sau la închiderea calculatorului).

Până acum am lucrat numai cu date citite de la tastatură, memorate în variabile simple. Pe parcursul execuţiei unui program, se alocă memorie în RAM pentru variabilele utilizate în programul respectiv. Memoria alocată se eliberează fie pe parcursul execuţiei programului, fie la terminarea execuţiei acestuia. În orice caz, după terminarea programului, valorile variabilelor „s-au pierdut", deoarece zona de memorie în care au fost memorate s-a eliberat şi, dacă doream să executăm încă o dată programul cu aceleaşi date de intrare, trebuia să le introducem din nou de la tastatură. Chiar şi pentru programe cu un volum mic de date de intrare, aşa cum am elaborat noi până acum, acest aspect reprezintă un inconvenient. Dar imaginaţi-vă ce s-ar întâmpla la o firmă care, de exemplu, la fiecare rulare a programului de salarii ar trebui să introducă toate datele despre angajaţii firmei sau la o bibliotecă la care, de exemplu, orice cerere de împrumut s-ar solda cu introducerea tuturor cărţilor existente în bibliotecă! Exemplele pot continua, dar ideea principală rămâne că în orice situaţie practică fişierele sunt indispensabile.

Fişierele pot fi clasificate după conţinutul lor astfel:
- fişiere *text*: conţin o secvenţă de caractere ASCII structurate pe linii;
- fişiere *binare*: conţin o secvenţă de octeţi, fără o structură predefinită.

Noi vom studia modul de utilizare a fişierelor text.

3.2. *Fişiere text în limbajul C++*

Declararea fişierelor

Operaţiile de intrare/ieşire în limbajul C++ au utilizat până acum *stream*-urile `cin` şi `cout`, care erau automat asociate tastaturii, respectiv ecranului. Pentru ca un program să poată citi informaţii dintr-un fişier, respectiv să scrie informaţii într-un fişier, trebuie să asociem fişierul respectiv unui *stream* de intrare/ieşire.

În fişierul antet `fstream.h` sunt declarate clasele `ifstream`, `ofstream` şi `fstream`[20]. Pentru a utiliza într-un program operaţii de intrare/ieşire folosind fişiere trebuie să declarăm variabile de tipurile `ifstream`, `ofstream` sau `fstream`. Mai exact:

`ifstream` – declarare *stream* de intrare (numai pentru operaţii de citire);
`ofstream` – declarare *stream* de ieşire (numai pentru operaţii de scriere);
`fstream` – declarare *stream* de intrare/ieşire (în care se pot realiza atât operaţii de citire, cât şi operaţii de scriere, în funcţie de modul specificat la deschidere).

Exemple

```
ifstream f1; //am declarat un stream de intrare denumit f1
ofstream f2; //am declarat un stream de iesire denumit f2
fstream  f3; //am declarat un stream denumit f3.
```

Observaţii

1. Declaraţiile precedente respectă sintaxa declaraţiilor de variabile (`ifstream`/`ofstream`/`fstream` reprezintă tipul, iar `f1`, `f2`, `f3` reprezintă numele variabilelor). Tipul fiind o clasă, variabilele `f1`, `f2`, `f3` sunt obiecte.
2. Nu intrăm în amănunte referitoare la programarea orientată pe obiect, dar trebuie să precizăm că o clasă este asemănătoare tipului `struct`, dar clasa conţine atât date, cât şi funcţiile necesare prelucrării acestora. Datele şi funcţiile incluse într-o clasă se numesc membri. Variabilele de tip clasă se numesc obiecte. Pentru a ne referi la un membru al unui obiect utilizăm operatorul punct, denumit operator de selecţie directă: `nume_obiect.nume_membru`.

Deschiderea fişierelor

Pentru a utiliza *stream*-ul declarat trebuie să îl deschidem. La deschidere, se asociază *stream*-ului un fişier fizic (existent pe suport extern) şi, eventual, se precizează modul de deschidere, care determină operaţiile permise cu fişierul respectiv. Deschiderea se poate realiza în două moduri:

20. La nivel de iniţiere, vom trata aceste clase ca un tip al limbajului C++.

- după declarare, prin apelul funcției membră `open`, precizând ca parametri un șir de caractere, care reprezintă specificatorul fișierului fizic, conform sintaxei sistemului de operare utilizat, și (eventual) modul de deschidere;
- la declarare, specificând după numele *stream*-ului care se declară numai parametrii corespunzători funcției `open`, încadrați între paranteze rotunde.

La deschidere, este obligatoriu să specificăm modul de deschidere dacă fișierul este declarat de tip `fstream`. Pentru tipul `ifstream` se utilizează implicit modul `ios::in`, iar pentru tipul `ofstream` se utilizează implicit modul `ios::out`.

Exemplul 1

```
ifstream f1;
f1.open("date.in");
```

Am declarat un *stream* de intrare denumit `f1`, apoi l-am deschis cu ajutorul funcției `open`, asociindu-l fișierului `date.in`. Deoarece nu am precizat discul logic pe care se află acest fișier sau calea către directorul în care se află fișierul, se deduce că fișierul se află pe hard-disc în directorul curent. Același efect l-am fi obținut prin deschiderea fișierului la declarare, astfel:

```
ifstream f1("date.in");
```

Exemplul 2

```
ofstream f2;
f2.open("c:\\ema\\exemple\\date.out");
```

Am declarat un *stream* de ieșire, denumit `f2`, pe care l-am asociat fișierului fizic `date.out`, care se află pe discul `c:`, în directorul `exemple`, care este un subdirector al directorului `ema`[21]. În acest exemplu nu am specificat nici un mod de deschidere. În acest caz se utilizează modul de deschidere implicit (se creează un fișier vid cu numele specificat, în care se vor realiza doar operații de scriere).

Exemplul 3

```
fstream f3;
f3.open("nr.in", ios::in);
```

Am declarat un *stream* denumit `f3`, pe care la deschidere l-am asociat fișierului `nr.in`. Deoarece tipul `fstream` nu are asociat un mod de deschidere implicit, a fost necesar să precizăm modul de deschidere. În acest caz, modul de deschidere este `ios::in`, ceea ce înseamnă că fișierul a fost deschis ca fișier de intrare (numai pentru operații de citire).

Exemplul 4

```
fstream f4;
f4.open("nr.out", ios::out);
```

21. Observați că în șirul de caractere care reprezintă specificatorul de fișier a fost necesar să dublăm caracterul special *backslash*!

Am declarat un *stream* denumit `f4`, pe care la deschidere l-am asociat fişierului `nr.out`. Modul de deschidere specificat este `ios::out`, ceea ce semnifică deschiderea fişierului ca fişier de ieşire (se creează un fişier vid cu numele specificat, în care se vor realiza doar operaţii de scriere).

Exemplul 5

```
fstream f5;
f4.open("nr.ad", ios::app);
```

Am declarat un *stream* denumit `f5`, pe care la deschidere l-am asociat fişierului `nr.ad`. Modul de deschidere specificat este `ios::app`, ceea ce înseamnă că deschidem fişierul ca fişier de ieşire, pentru adăugare de informaţii la sfârşit[22].

Observaţie

Dacă după operaţia de deschidere a fişierului variabila corespunzătoare are valoarea `NULL`, deducem că operaţia de deschidere a eşuat (de exemplu, dacă dorim să deschidem un fişier de intrare care nu există). Este indicat să testăm succesul operaţiei de deschidere înainte de a efectua orice altă operaţie cu fişierul.

Citirea datelor dintr-un fişier

După deschiderea fişierului ca fişier de intrare se pot realiza operaţii de citire. În acest scop se poate utiliza operatorul de citire >>, sau pot fi utilizate funcţii-membru specifice.

Una dintre particularităţile citirii cu operatorul >> este faptul că ignoră caracterele albe. Prin urmare, dacă intenţionăm să citim toate caracterele din fişier (inclusiv spaţiu, tab, enter), acest mod de citire este inadecvat.

În acest scop putem utiliza funcţia-membru `get()`. Funcţia `get()` are mai multe forme de apel. De exemplu, pentru citirea caracterelor (`char` sau `unsigned char`) se poate utiliza astfel:

```
ifstream f1("text.in");
char c;
f1.get(c);
```

Am citit primul caracter din fişier, indiferent dacă este sau nu caracter alb. Evident, în mod similar, funcţia `get()` se poate utiliza pentru citirea unui caracter de la tastatură: `cin.get(c)`.

Scrierea datelor într-un fişier

După deschiderea fişierului ca fişier de ieşire, se pot realiza operaţii de scriere utilizând operatorul de scriere <<.

22. app – append (adăugare).

Exemplu

```
ofstream fout("date.out");
fout<<"Salut";
```

Operații de test

Detectarea sfârșitului de fișier

În multe aplicații este necesar să citim toate datele existente într-un fișier, fără să avem informații prealabile despre numărul acestora.

La deschiderea unui fișier se creează un *pointer de fișier*, care indică poziția curentă în *stream*. Orice operație de citire determină deplasarea pointerului de citire în *stream*-ul de intrare, respectiv orice operație de scriere determină deplasarea pointerului de scriere în *stream*-ul de ieșire.

Pentru a testa dacă pointerul de fișier a ajuns la sfârșitul fișierului putem utiliza funcția membră[23] `eof()`. Funcția `eof()` returnează valoarea `0` dacă nu am ajuns la sfârșitul fișierului, respectiv o valoare diferită de `0` dacă am ajuns la sfârșit.

Exemplu

Vom citi și vom număra toate caracterele existente într-un fișier, până la sfârșitul acestuia.

```
ifstream fin("text.in");
char c;
long int nr=0;
while (!fin.eof()) {fin.get(c); nr++;}
cout<<"Fisierul contine "<<nr<<" caractere\n";
```

Funcția membru `get()` returnează pointerul de fișier (poziția curentă în *stream*). Prin urmare, secvența de program ar putea fi rescrisă și astfel:

```
ifstream fin("text.in");
char c;
long int nr=0;
while (fin.get(c)) nr++;
cout<<"Fisierul contine "<<nr<<" caractere\n";
```

Testarea reușitei unei operații de intrare/ieșire

În aplicațiile didactice, de cele mai multe ori presupunem că datele de intrare sunt corecte. În condiții practice însă trebuie să verificăm reușita oricărei operații de intrare/ieșire și, mai mult, să verificăm și corectitudinea datelor citite.

Pentru a testa reușita unei operații de intrare/ieșire se poate utiliza, de exemplu, funcția membră `good()`. Funcția `good()` returnează o valoare diferită de `0` dacă operația precedentă de intrare/ieșire s-a efectuat cu succes și `0` în caz contrar.

23. Denumirea *eof* provine de la *end of file* – sfârșit de fișier.

Există şi alte funcţii de test asemănătoare: `bad()` (care returnează o valoare diferită de `0` dacă s-a efectuat o operaţie incorectă) sau `fail()` (care returnează o valoare diferită de `0` dacă a apărut o eroare în operaţiile de intrare/ieşire până în acel moment).

Anticiparea următorului caracter la intrare

În anumite situaţii este necesară citirea unor date până la întâlnirea unui anumit caracter, cu excepţia acestuia. Pentru a afla anticipat ce caracter urmează în *stream*-ul de intrare (fără a-l extrage din *stream*) putem utiliza funcţia membră `peek()`, care returnează următorul caracter din *stream*.

Închiderea unui fişier

După realizarea tuturor operaţiilor cu un fişier, acesta trebuie închis. Închiderea unui fişier se realizează prin apelarea funcţiei membre `close()`.

Exemplu

```
f1.close();
```

Operaţia de închidere este obligatorie, în special pentru fişierele de ieşire. *În cazul în care nu închideţi fişierul de ieşire, există mari şanse să pierdeţi informaţii!*

3.3. *Fişiere text în limbajul C*

Declararea fişierelor

Operaţiile de intrare/ieşire în limbajul C au fost realizate până acum utilizând instrucţiunile `scanf()` şi `printf()`, care implicit preiau informaţii de la tastatură şi, respectiv, le transmit spre ecran. Pentru ca un program să poată prelucra informaţii dintr-un fişier, acesta trebuie declarat, indicându-se numele lui precum şi operaţiile permise asupra conţinutului fişierului respectiv.

În fişierul antet `stdio.h` sunt declarate, printre altele, şi funcţiile, constantele, tipurile de date şi variabilele globale cu ajutorul cărora pot fi prelucrate informaţiile conţinute în fişiere. Pentru a utiliza într-un program operaţii de intrare/ieşire folosind fişiere trebuie să declarăm una sau mai multe variabile de tip `FILE *`. Declaraţia unui fişier se face astfel:

```
FILE * fisier;
```

Observaţie

Declaraţia precedentă respectă sintaxa declaraţiilor de variabile (`FILE *` reprezintă tipul, iar `fisier` reprezintă numele variabilei).

Deschiderea fişierelor

Desigur, pentru a avea acces la informaţiile din fişier trebuie să fie cunoscut fişierul fizic de pe suport (specificatorul de fişier conform sintaxei sistemului de operare utilizat), ca şi modul în care vor fi prelucrate informaţiile din fişier. Acest lucru se realizează utilizând funcţia `fopen`.

```
fisier = fopen(const char *nume, const char *mod)
```

unde `nume` este o constantă de tip şir de caractere care reprezintă specificatorul fişierului fizic pe suport, iar `mod` este o constantă şir de caractere care indică modul în care vor fi accesate informaţiile din fişier.

Exemple

```
FILE *f1, *f2, *f3;   //am declarat trei fisiere f1, f2, f3
f1 = fopen("date.in", "r");
   //am deschis fisierul de intrare (r - read) f1
f2 = fopen("date.out", "w");
   //am deschis fisierul de iesire (w - write) f2
f3 = fopen("date.txt", "a");
   //am deschis fisierul f3 pentru adaugare de
   //informatii (a - append)
```

Declararea şi deschiderea unui fişier se pot face şi simultan:

Exemplu 1

```
FILE *f1 = fopen("date.in", "r");
```

Am declarat fişierul de intrare denumit `f1`, apoi l-am deschis cu ajutorul funcţiei `fopen`, asociindu-l fişierului `date.in`. Deoarece în toate exemplele anterioare nu am precizat discul logic pe care se află fişierele sau calea către directorul în care se află acestea, se deduce că fişierele se află pe hard-disc în directorul curent.

Exemplul 2

```
FILE *f2;
f2 = fopen("c:\\ema\\exemple\\date.out", "w");
```

Am declarat un fişierul de ieşire, denumit `f2`, pe care l-am asociat fişierului fizic `date.out`, care se află pe discul `c:`, în directorul `exemple`, care este un subdirector al directorului `ema`.

Observaţie

Dacă, după operaţia de deschidere a fişierului, variabila corespunzătoare are valoarea `NULL`, deducem că operaţia de deschidere a eşuat (de exemplu, dacă dorim să deschidem un fişier de intrare care nu există). Este indicat să testăm succesul operaţiei de deschidere înainte de a efectua orice altă operaţie cu fişierul.

Exemplul 3

```
#include <stdio.h>
int main()
{ FILE *numef;
  if (!(numef = fopen("nr.in", "r")))
     printf("EROARE la deschidere\n");
     else printf("Deschidere corecta a fisierului\n");
  return 0; }
```

Testarea erorii la deschiderea fişierului se poate face şi astfel:

```
if ((numef = fopen("nr.in", "r")) == NULL)
```

Citirea datelor dintr-un fişier

După deschiderea fişierului ca fişier de intrare se pot realiza operaţii de citire. În acest scop se utilizează funcţia `fscanf`.

```
int fscanf(fisier, format, adr_var1, ..., adr_varn);
```

Observaţi că prototipul funcţiei `fscanf()` este asemănător cu prototipul funcţiei `scanf()`. Funcţia `fscanf()` are în plus, ca prim parametru, fişierul din care se face citirea. În rest, semnificaţia parametrilor, rezultatului şi efectul funcţiei sunt aceleaşi ca în cazul funcţiei `scanf()`.

Scrierea datelor într-un fişier

După deschiderea fişierului ca fişier de ieşire, se pot realiza operaţii de scriere utilizând funcţia `fprintf()`.

```
int fprintf(fisier, format, expresie1, ..., expresien);
```

Observaţi că prototipul funcţiei `fprintf()` este asemănător cu prototipul funcţiei `printf()`. Funcţia `fprintf()` are în plus, ca prim parametru, fişierul în care se face afişarea. În rest, semnificaţia parametrilor, rezultatului şi efectul funcţiei sunt aceleaşi ca în cazul funcţiei `printf()`.

Închiderea fişierelor

După terminarea tuturor operaţiilor care implică lucrul cu fişiere, acestea trebuie închise. Funcţia care realizează acest lucru este:

```
fclose(fisier);
```

Exemplul 1

Să scriem un program care citeşte din fişierul de intrare `nr.in` un număr natural `n` (`0<n≤100`) şi apoi `n` valori naturale şi afişează în fişierul de ieşire `nr.out` numerele citite descompuse în factori primi, pe fiecare linie fiind scris câte un număr urmat de perechi de forma `(d,p)`, indicând divizorul şi puterea lui.

```
#include <stdio.h>

int main()
{ int n, i, d, p, er;
  unsigned long x;
  FILE *fin = fopen("nr.in", "r");
  if (!fin)
    {printf("Eroare la deschiderea fisierului nr.in\n");
     return 1;}
  FILE *fout = fopen("nr.out", "w");
  er = fscanf(fin, "%d", &n); //citesc n din fisierul fin
  if (!er || n>100 || n<0)
    {printf("Numarul de valori este incorect\n");
     return 2;}
  for (i=0; i<n; i++)     //citesc cele n valori
    { er = fscanf(fin, "%lu", &x);
      if (!er)
        { printf("Eroare la citirea valorilor\n");
          return 3;}
      fprintf(fout, "%10lu ", x);
      d=2;
      while (x>1)
        { p=0;
          while (x%d==0)
            { p++; x/=d; }
          if (p) fprintf(fout, "(%d,%d)", d, p);
          d++;}
      fprintf(fout, "\n");
    }
  fclose(fout);
  return 0;
}
```

Exemplul 2

Scrieți un program care creează o copie a fișierului `intrare.txt` cu numele `iesire.txt`.

```
#include <stdio.h>

int main()
{
FILE *fisin, *fisout;
char x;
if (!(fisin = fopen("intrare.txt", "r")))
   {printf("EROARE la deschiderea fisierului de intrare");
     return 1;}
if (!(fisout = fopen("iesire.txt", "w")))
   {printf("EROARE la deschiderea fisierului de iesire");
    return 2;}
```

```
while (!feof(fisin))
      if (fscanf(fisin, "%c", &x) != EOF)
         fprintf(fisout, "%c", x);
 fclose(fisin);
 fclose(fisout);}
 return 0; }
```

3.4. *Probleme propuse*

1. Ce va conţine fişierul `test.out` pentru `n=31` şi `a=3`? Care credeţi că este efectul programului?

```
#include <fstream.h>
#include <iomanip.h>
int main()
{
ofstream fout("test.out");
int a, n, i, j;
cout<<"n (28<=n<=31), a (1<=a<=7) = "; cin >> n >> a;
for (i=1; i<= 7; i++)
     {j=i-a+1;
      if (j>0)  fout<<setw(4)<<j;  else fout << "    ";
      for (j += 7; j <= n; j+=7) fout<<setw(4)<<j;
      fout << endl;}
fout.close();
return 0;
}
```

2. Fişierul `nr.in` conţine numere întregi, scrise pe mai multe linii, numerele de pe aceeaşi linie fiind separate prin spaţii. Scrieţi un program care să numere câte valori întregi sunt pe fiecare dintre liniile fişierului.
3. Scrieţi un program care să testeze dacă două fişiere ale căror nume se citesc de la tastatură sunt identice.
4. În fişierul `NR.TXT` se află numere naturale din intervalul `[0,5000]`, separate prin spaţii. Să se creeze fişierul `PARE.TXT` care să conţină doar valorile pare din fişierul `NR.TXT`, câte o valoare pe linie.

(variantă Bacalaureat, 2000)

5. Scrieţi un program care să numere câte linii conţine fişierul text `text.in`.

4. Tipuri structurate de date

Pentru început, am scris programe care nu prelucrează decât date simple (de exemplu, numere întregi, numere reale, caractere). În realitate, un program trebuie să prelucreze volume mari de date şi pentru ca prelucrarea să se realizeze eficient este *necesară* organizarea acestor date în structuri. De exemplu, să presupunem că dorim să ordonăm elevii din şcoală în ordine alfabetică. Pentru aceasta este nevoie să reţinem „undeva" (într-o structură de date) numele şi prenumele elevilor şcolii şi abia apoi îi putem ordona.

O *structură de date* reprezintă un ansamblu (o colecţie) de date organizate după anumite reguli, reguli care depind de tipul de structură.

4.1. *Tablouri*

Un *tablou* este o colecţie de date *de acelaşi tip*, memorate într-o zonă de *memorie contiguă*, reunite sub un nume comun (numele tabloului).

Declararea unei variabile de tip tablou:

```
tip nume[NrE];
```

Am declarat un tablou format din `NrE` elemente de tipul `tip`. `NrE` indică numărul de elemente din tablou şi trebuie să fie obligatoriu o expresie constantă.

Deoarece elementele unui tablou sunt memorate în ordine, unul după altul, într-o zonă contiguă, pentru a ne referi la un element al unui tablou putem specifica numele tabloului din care face parte elementul şi poziţia sa în tablou, prin numărul său de ordine (numerotarea începe de la `0`).

```
nume[indice]
```

Am specificat elementul `indice` al tabloului `nume` (`indice` reprezintă numărul de ordine al elementului în tablou, cuprins între `0` şi `NrE-1`). Parantezele pătrate (`[]`) constituie *operatorul de indexare*. Operatorul de indexare are prioritate maximă (mai mare decât a operatorilor unari).

Exemple

1. Să declarăm un tablou cu 10 elemente de tip int:

```
int a[10];
```

Elementele tabloului sunt a[0], a[1], a[2], ..., a[9].

2. Să declarăm un tablou cu 100 de elemente de tip float:

```
float b[100];
```

Observații

1. La întâlnirea unei declarații de variabilă tablou, compilatorul verifică dacă dimensiunea zonei de memorie necesară pentru memorarea tabloului nu depășește memoria disponibilă. Dimensiunea zonei de memorie necesară unui tablou se calculează înmulțind numărul de elemente cu numărul de octeți necesari pentru memorarea unui element, adică NrE*sizeof(tip).
2. Ca în cazul oricărei declarații de variabile, putem inițializa elementele unei variabile tablou, chiar de la declarare.

```
tip nume[NrE]={val0, val1, ..., valk};
```

Ca urmare, se vor atribui în ordine elementelor tabloului valorile din lista de inițializare (k≤NrE). Dacă tabloul este integral inițializat la declarare, nu este necesar să mai specificăm dimensiunea sa, fiind considerată egală cu numărul de valori din lista de inițializare. De exemplu:

```
int a[]={12,20,30,5};
```

Am declarat un tablou cu 4 elemente de tip int inițializate astfel:

12	20	30	5
a[0]	a[1]	a[2]	a[3]

3. Un astfel de tablou, pentru care la declarare este specificată o singură dimensiune, iar poziția unui element este specificată utilizând un singur indice, se numește *tablou unidimensional* sau *vector*.
4. Elementele unui tablou pot fi de orice tip al limbajului. Prin urmare... elementele unui tablou pot fi de tip tablou! Declarare:

```
tip nume[Nr1][Nr2];
```

Am declarat un tablou cu Nr2 elemente, fiecare element fiind un tablou cu Nr1 elemente de tipul specificat. Un astfel de tablou, pentru care la declarare trebuie să specificăm două dimensiuni, iar poziția unui element este specificată utilizând doi indici, se numește *tablou bidimensional* sau *matrice*.

Putem să ne imaginăm un tablou bidimensional ca pe o tablă de șah. Poziția unui element pe tablă este identificată prin doi indici: linia și coloana. Prin analogie cu tabla de șah, și la informatică primul indice utilizat în referirea unui element este denumit „indice de linie", iar cel de al doilea indice este denumit „indice de coloană".

De exemplu, să declarăm o matrice cu două linii și trei coloane cu elemente întregi, pe care o vom inițializa la declarare:

```
int a[2][3]={{1,2,3},{4,5,6}};
```

a	coloana 0	coloana 1	coloana 2
linia 0	1	2	3
linia 1	4	5	6

Pentru a ne referi la un element al unei matrice, specificăm numele matricei, indicele de linie şi indicele de coloană astfel:

```
nume[indice_linie][indice_coloana]
```

De exemplu, pentru a ne referi la elementul de pe linia 1, coloana 2 din matricea a, vom scrie a[1][2].

5. Elementele unui tablou bidimensional pot fi de orice tip, inclusiv... tablou! Se obţin astfel tablouri multidimensionale. Să declarăm, de exemplu, un tablou tridimensional (vă imaginaţi un paralelipiped cu dimensiunile 10, 5, 40):

```
int a[10][5][40];
```

4.2. *Prelucrări elementare pe vectori*

Citirea unui vector

Citirea unui vector se realizează citind elementele vectorului, unul câte unul. Să considerăm ca exemplu un vector a cu n elemente (n≤100) de tip int.

```
int a[100], n, i;          //declarare vector
cout << "n= "; cin >>n;    //citesc numarul de componente
for (i=0; i<n; i++)        //citesc succesiv componentele
     {cout <<"a["<<i<<"]="; cin>>a[i];}
```

Afişarea unui vector

Afişarea unui vector se realizează afişând pe rând componentele vectorului.

```
for (i=0; i<n; i++)        //afisez succesiv componentele
     cout << a[i] <<' ';
```

Copierea unui vector

Să presupunem că dorim să copiem vectorul a într-un alt vector b. Copierea unui vector nu se poate face printr-o atribuire de forma b=a (veţi obţine un mesaj de eroare). Copierea unui vector se realizează element cu element:

```
for (i=0; i<n; i++) b[i]=a[i];
```

Determinarea elementului maxim/minim dintr-un vector

Pentru a determina cel mai mare element din vector, vom considera o variabilă (să o numim max) în care vom reţine la fiecare pas maximul dintre elementele analizate. Iniţializăm variabila max cu un element din vector (de exemplu, cu

a[0]). Parcurgem apoi vectorul, comparând fiecare element din vector cu max şi actualizând eventual maximul după comparare.

```
max=a[0];              //initializez maximul cu primul element
for (i=1; i<n; i++) //compar succesiv componentele cu max
     if (max<a[i]) max=a[i];              //actualizez max
```

Exerciţii

1. Modificaţi secvenţa de instrucţiuni precedentă, astfel încât să determine cel mai mic element din vector.
2. Modificaţi secvenţa de instrucţiuni precedentă, astfel încât să determine maximul, după iniţializarea lui max cu ultimul element din vector.

Media aritmetică a elementelor strict pozitive

Se consideră un tablou de 10 numere întregi. Scrieţi un program care citeşte de la tastatură cele 10 componente ale vectorului şi afişează pe ecran media aritmetică a valorilor strict pozitive din vector, cu două zecimale.

(variantă Bacalaureat, 2000)

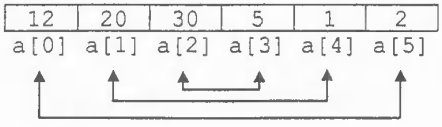
Soluţie

După citire, se parcurge vectorul, verificând pentru fiecare element dacă este sau nu strict pozitiv. Când găsim un element strict pozitiv, îl numărăm şi îl adunăm la suma elementelor strict pozitive:

```
#include <iomanip.h>
void main()
{ float a[10], s=0;
  int i, nr=0;
  for (i=0; i<10;i++) cin >> a[i];
  for (i=0; i<10; i++)
      if (a[i]>0) s+=a[i], nr++;
  if (nr) cout << setprecision(2) << s/nr;
      else cout << "Nu exista elemente strict pozitive";}
```

Inversarea ordinii elementelor din vector

Pentru a inversa ordinea elementelor dintr-un vector trebuie ca primul element să fie interschimbat cu ultimul, al doilea element cu penultimul ş.a.m.d. Mai exact, trebuie să parcurgem vectorul până la mijloc şi să interschimbăm elementele simetrice faţă de mijloc. De exemplu:



Problema care rămâne este determinarea simetricului.

Poziția elementului	Poziția simetricului	Observație
0	n-1	0+n-1=n-1
1	n-2	1+n-2=n-1
2	n-3	2+n-3=n-1

...

Observăm că atunci când poziția elementului crește cu 1, poziția simetricului scade cu 1, deci suma dintre poziția elementului și poziția simetricului este constantă (n-1). Deducem că simetricul elementului a[i] este a[n-1-i].

```
for (i=0; i<n/2; i++)
    {aux=a[i]; a[i]=a[n-1-i]; a[n-i-1]=aux;}
```

Verificarea unei proprietăți

Frecvent apar probleme în care se cere să se verifice dacă toate elementele unui vector au o anumită proprietate *P* sau dacă există în vector un element care are proprietatea *P*. Pentru a descrie o secvență de instrucțiuni cât mai generală, considerăm că P(x) are valoarea 1, dacă x are proprietatea respectivă, și 0, altfel.

a. Pentru a verifica dacă toate elementele unui vector au proprietatea *P*, vom verifica succesiv fiecare element din vector. Pentru a reține rezultatul verificării, vom considera o variabilă întreagă ok, care va avea valoarea 1, dacă toate elementele vectorului au proprietatea *P*, și 0, altfel. Inițial variabilei ok îi atribuim valoarea 1 (ipoteza optimistă, „prezumția de nevinovăție" – din moment ce nu am găsit încă nici un element care să conteste acest lucru, putem presupune că toate elementele vectorului au proprietatea *P*). Dacă, parcurgând vectorul, vom găsi un element care nu are proprietatea respectivă, variabilei ok îi vom atribui valoarea 0:

```
for (ok=1, i=0; i<n && ok; i++)
//parcurg vectorul pana la sfarsit sau pana la intalnirea
//unui element care nu are proprietatea P
     if ( !P(a[i]) ) ok=0; //a[i] nu are proprietatea P
```

b. Pentru a verifica dacă există un element în vector cu proprietatea *P*, vom parcurge vectorul (până la sfârșit sau până la întâlnirea unui element cu proprietatea *P*). Rezultatul verificării îl vom reține în variabila întreagă gasit. Inițial (deoarece nu am găsit încă nici un element cu proprietatea *P*) îi atribuim variabilei gasit valoarea 0. Dacă vom găsi în vector un element cu proprietatea *P*, îi vom atribui variabilei gasit valoarea 1:

```
for (gasit=i=0; i<n && !gasit; i++)
     if ( P(a[i]) ) gasit=1; //a[i] are proprietatea P
```

Căutarea unui element într-un vector

Fie a un vector cu n (n≤100) componente întregi și x o valoare întreagă. Verificați dacă x apare sau nu în vectorul a.

Practic, trebuie să verificăm dacă există în vectorul a un element cu proprietatea că este egal cu x. Particularizând secvența de verificare de la punctul b, obținem:

```
for (gasit=i=0; i<n && !gasit; i++)
     if (a[i]==x) gasit=1;  //am gasit valoarea x in vector
```

Această metodă de căutare, în care testăm succesiv elementele vectorului, se numește *căutare secvențială*. În cazul cel mai defavorabil (când x nu se găsește în vector sau este plasat pe ultima poziție), căutarea secvențială efectuează n comparații. O soluție mai eficientă nu există, în cazul general.

În realitate ne confruntăm frecvent cu problema căutării unui element într-o mulțime și frecvent mulțimea respectivă este ordonată (de exemplu, căutarea unui cuvânt în dicționar, a unui număr în cartea de telefon etc.). În astfel de situații nu aplicăm o căutare secvențială (de exemplu, pentru a găsi în cartea de telefon numărul lui Popescu Ion nu începem să căutăm de la litera 'A').

Cum procedăm? Deschidem cartea de telefon la întâmplare. Verificăm dacă am dat peste Popescu Ion. Dacă nu, verificăm dacă Popescu Ion este în prima parte a cărții sau în cea de a doua și continuăm căutarea numai în porțiunea respectivă. Procedeul se repetă până când găsim sau până când nu mai avem unde căuta.

Aceeași idee poate fi aplicată și pentru căutarea unui element într-un vector ordonat. Cum pentru calculator este dificil să lucreze „la întâmplare", vom lucra prin înjumătățiri succesive: mai întâi comparăm elementul căutat cu elementul din mijloc. Dacă este egal, am găsit, am terminat. Dacă nu este egal, verificăm dacă elementul căutat este mai mare decât elementul din mijloc (în acest caz, căutăm mai departe numai în cea de-a doua jumătate). Dacă elementul căutat este mai mic decât elementul din mijloc, căutăm mai departe numai în prima jumătate.

```
for (st=0, dr=n-1, gasit=0; !gasit && st<=dr;)
//cautam pe x de la pozitia st la pozitia dr; cautarea
//continua pana gasim sau pana nu mai avem unde cauta
    { mijloc=(st+dr)/2;                  //calculam mijlocul
      if (a[mijloc]==x) gasit =1;        //am gasit pe x
         else
         if (a[mijloc]<x) st=mijloc+1; //caut in stanga
             else dr=mijloc-1;}         //caut in dreapta
if (gasit) cout<<x<<" se gaseste pe pozitia "<<mijloc;
    else cout <<x<<" nu se afla in vector ";
```

Acest algoritm de căutare se numește *căutare binară*. Denumirea este sugestivă: la fiecare pas, alegem una din cele două alternative posibile: caut la stânga elementului din mijloc sau în dreapta lui.

Exercițiu

Să considerăm un vector cu n=7 elemente ordonate crescător. Câte comparații execută algoritmul de căutare binară în cazul cel mai defavorabil (când elementul căutat nu se află în vector sau este depistat abia la ultima comparație)? Dar pentru n=10 elemente? Puteți estima în general (în funcție de n) numărul de comparații necesare în cazul cel mai defavorabil? Ce concluzie trageți, comparând algoritmul de căutare binară cu algoritmul de căutare secvențială?

Sortare

Fie n ($n \in \mathbf{N}^*$) elemente $a_0, a_1, \ldots, a_{n-1}$ dintr-o mulțime total ordonată. Ordonați crescător elementele $a_0, a_1, \ldots, a_{n-1}$.

Problema ordonării unor elemente (cunoscută și sub denumirea de sortare) este frecvent întâlnită în practică și din acest motiv a fost studiată intens. Ca urmare, au fost elaborați numeroși algoritmi de sortare. Cum, de obicei, numărul de elemente care trebuie să fie ordonate este mare, s-a studiat și eficiența acestor algoritmi, în scopul elaborării unor algoritmi de sortare performanți. Pentru început, vom studia algoritmi de sortare simpli, nu performanți, urmând ca ulterior să învățăm să evaluăm eficiența acestor algoritmi și să elaborăm algoritmi eficienți.

Sortare prin selecție

Sortarea prin selecție are două variante: sortarea prin selecția elementului maxim și sortarea prin selecția elementului minim. În ambele variante, ideea de bază este aceeași: se selectează cel mai mare element din vector (sau cel mai mic) și se plasează pe ultima poziție în vector (respectiv, pe prima poziție). Apoi se calculează cel mai mare dintre elementele rămase și se plasează pe penultima poziție în vector (sau cel mai mic dintre elementele rămase și se plasează pe a doua poziție) ș.a.m.d. Acest procedeu se repetă de n-1 ori.

```
for (dr=n-1; dr>0; dr--)    //calculez maximul de la 0 la dr
    {for (max=a[0],pozmax=0,i=1; i<=dr; i++)
          if (a[i] > max)  max=a[i], pozmax=i;
     a[pozmax]=a[dr];       //plasez maximul pe pozitia dr
     a[dr] = max; }
```

Observație

La fiecare iterație a ciclului for exterior este calculat $\max\{a_0, a_1, \ldots, a_{dr}\}$ și este plasat pe poziția dr, elementele de la dr+1 la n-1 fiind deja plasate pe pozițiile lor definitive. Pentru a calcula $\max\{a_0, a_1, \ldots, a_{dr}\}$ sunt necesare dr comparații. Deci, în total se execută $1+2+...+n-1=n\cdot(n-1)/2$ comparații, indiferent de ordinea inițială a elementelor vectorului.

Exerciţii

1. Modificaţi algoritmul precedent, astfel încât să realizeze ordonarea descrescătoare a elementelor vectorului.
2. Modificaţi algoritmul precedent, astfel încât să realizeze ordonarea prin selecţia minimului.

*Sortare prin compararea vecinilor (*bubblesort*)*

O altă metodă de sortare este de a parcurge vectorul, comparând fiecare două elemente vecine şi (dacă este cazul) interschimbându-le. Cum într-o singură trecere nu se poate realiza sortarea vectorului, acest procedeu se repetă până când vectorul devine sortat (la ultima trecere nu am mai efectuat nici o interschimbare).

```
do
  {schimb=0;          //initial nu am facut nici o schimbare
   for (i=0; i<n-1; i++)                //parcurg vectorul
        if (a[i]>a[i+1])      //compar doua elemente vecine
           { //nu sunt in ordine, le interschimb
             aux=a[i]; a[i]=a[i+1]; a[i+1]=aux;
             schimb=1;}       //retin ca am facut schimbari
  }     //procedeul se repeta cat timp se executa schimbari
while (schimb);
```

Observaţii

1. Metoda este denumită şi *bubblesort* (metoda bulelor) deoarece la fiecare nouă trecere prin vector elementele cu valori mari migrează către sfârşitul vectorului („urcă" în vector, aşa cum bulele de aer se ridică la suprafaţă când fierbe apa).
2. În cazul cel mai defavorabil (când valorile sunt ordonate descrescător), această metodă execută aproximativ n^2 operaţii.

Exerciţii

1. Modificaţi algoritmul precedent, astfel încât să realizeze ordonarea descrescătoare a elementelor vectorului.
2. Observăm că, dacă la parcurgerea curentă ultima interschimbare a fost efectuată la poziţia `x`, în oricare dintre parcurgerile ulterioare nu se vor efectua interschimbări după poziţia `x`. Îmbunătăţiţi algoritmul precedent utilizând această observaţie.

Sortare prin inserţie

Metoda de sortare prin inserţie este, de asemenea, o metodă simplă, pe care o utilizăm adesea când ordonăm cărţile la jocuri de cărţi: de fiecare dată când tragem o carte, o plasăm pe poziţia sa corectă, astfel încât în mână cărţile să fie ordonate.

Utilizând această idee, sortăm vectorul astfel: parcurgem vectorul, element cu element; la fiecare pas `i`, căutăm poziţia corectă a elementului curent `a[i]`, astfel încât secvenţa `a[0]`, `a[1]`, ..., `a[i]` să fie ordonată:

```
for (i=1; i<n; i++)
    {v=a[i];                    //caut pozitia lui v
     for (poz=i; poz && a[poz-1]>v; poz--)
         a[poz]=a[poz-1];    //mut la dreapta elementele > v
     a[poz] = v;  }      //poz este pozitia corecta pentru v
```

Exerciţiu

Modificaţi algoritmul precedent, astfel încât să realizeze ordonarea descrescătoare a elementelor vectorului.

Sortare prin numărarea aparițiilor

În anumite probleme, elementele vectorului au ca valori numere naturale dintr-un interval [0,Max) de dimensiune redusă (sau au valori care pot fi asociate numerelor naturale dintr-un astfel de interval). Prin dimensiune redusă înţelegem că se poate declara un vector V cu Max componente întregi.

În acest caz particular, cea mai bună metodă este sortarea prin numărarea aparițiilor: se numără pentru fiecare valoare din intervalul [0, Max) de câte ori apare în vector.

```
#include <iostream.h>
#define Max 1000
#define NrMax 10000

int n, V[Max], a[NrMax];
int main()
{int x, i, j, nr;
 cout<<"n="; cin>>n;
 for (i=0; i<n; i++)
     {cin>>x;  //citesc o noua valoare
     V[x]++;} //marim numarul de aparitii ale valorii x
//plasam in ordine elementele in vectorul a
for (nr=i=0; i<Max; i++)
    for (j=0; j<V[i]; j++)
        a[nr++]=i;
for (i=0; i<n; i++) cout<<a[i]<<' ';
cout<<endl; return 0;}
```

Observaţie

Numărul de operaţii efectuate de acest algoritm de sortare este de ordinul lui n+Max.

Interclasare

Fie a un vector cu n elemente şi b un vector cu m elemente, ordonaţi crescător. Să se construiască un al treilea vector, c, care să conţină atât elementele vectorului a, cât şi elementele vectorului b, în ordine crescătoare.

O soluție (ineficientă, care nu ține cont de faptul că a și b sunt deja ordonați) este să copiem în vectorul c elementele din a și din b și apoi să sortăm vectorul c.

O altă idee este de a parcurge simultan cei doi vectori, comparând la fiecare pas elementul curent din a cu elementul curent din b. Cel mai mic dintre cele două elemente va fi copiat în vectorul c, avansând doar în vectorul din care am copiat. Când am epuizat elementele din unul din cei doi vectori, copiem elementele rămase în celălalt, deoarece nu mai avem cu ce le compara.

```
for (i=j=k=0; i<n && j<m;)
    if (a[i]<b[j])    //copiez in c elementul cel mai mic
       c[k++]=a[i++];                          //copiez din a
       else c[k++]=b[j++];                     //copiez din b
//copiez eventualele elemente ramase in a
for (; i<n; i++)
    c[k++]=a[i];
//copiez eventualele elemente ramase in b
for (; j<m; j++)
    c[k++]=b[j];
```

Exercițiu

Estimați în funcție de n și m numărul de comparații necesare pentru interclasare, în cazul cel mai defavorabil.

4.3. *Prelucrări elementare cu matrice*

Citirea unei matrice

Citirea unei matrice se realizează citind elementele matricei, linie cu linie. Să considerăm ca exemplu o matrice a cu n linii și m coloane (n,m≤100), cu elemente de tip int.

```
int a[100][100], n, m, i, j;          //declarare matrice
cout << "n, m= "; cin >> n >> m;
for (i=0; i<n; i++)          //parcurg matricea linie cu linie
    for (j=0; j<m; j++)    //citesc elementele liniei i
        {cout <<"a["<< i << "][" <<j << "]="; cin>>a[i][j];}
```

Afișarea unei matrice

Afișarea unei matrice se realizează afișând elementele matricei, linie cu linie:

```
for (i=0; i<n; i++)      //parcurg matricea linie cu linie
   {for (j=0; j<m; j++) //afisez componentele de pe linia i
      cout <<a[i][j]<<' ';
    cout<<endl;}         //trec la linia urmatoare
```

Parcurgerea unei matrice pe linii

Unele probleme impun parcurgerea unei matrice linie cu linie şi prelucrarea separată a elementelor de pe fiecare linie în parte. De exemplu, să calculăm suma elementelor de pe fiecare linie a unei matrice:

```
for (i=0; i<n; i++)
    {for (s=0, j=0; j<m; j++) s+=a[i][j];
     cout<<"Suma de pe linia "<<i<<" : " << s <<'\n';}
```

Parcurgerea unei matrice pe coloane

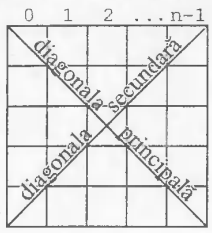
Există probleme care impun parcurgerea unei matrice pe coloane şi prelucrarea separată a elementelor de pe fiecare coloană în parte. De exemplu, să calculăm suma elementelor de pe fiecare coloană a unei matrice:

```
for (j=0; j<m; i++)
    {for (s=0, i=0; i<n; j++) s+=a[i][j];
     cout<<"Suma de pe coloana "<<j<<" :" <<s<<endl;}
```

4.4. *Prelucrări elementare specifice matricelor pătratice*

O matrice se numeşte *pătratică* dacă numărul de linii este egal cu numărul de coloane ale matricei.

Datorită acestei particularităţi, se pot realiza prelucrări specifice matricelor pătratice, de exemplu: parcurgerea diagonalelor sau a elementelor de sub sau de deasupra unei diagonale.



Parcurgerea diagonalelor

O matrice pătratică are două diagonale: diagonala principală (care uneşte primul element al matricei, `a[0][0]`, cu ultimul element, `a[n-1][n-1]`) şi diagonala secundară (care uneşte ultimul element de pe prima linie, `a[0][n-1]`, cu primul element de pe ultima linie, `a[0][n-1]`).

Prin urmare, elementele de pe diagonala principală sunt `a[0][0]`, `a[1][1]`, `a[2][2]`, ..., `a[n-1][n-1]`.

Pe diagonala principală, indicele de linie este egal cu indicele de coloană.

Afişarea elementelor de pe diagonala principală se poate realiza astfel:

```
for (i=0; i<n; i++) cout <<a[i][i]<<' ';
```

Elementele de pe diagonala secundară sunt `a[0][n-1]`, `a[1][n-2]`, `a[2][n-3]`, ..., `a[n-1][0]`. Observăm că pe diagonala secundară când indicele de linie creşte cu 1, indicele de coloană scade cu 1.

Pe diagonala secundară, suma dintre indicele de linie şi indicele de coloană este constantă:

`linie+coloana=constant=n-1⇒coloana=n-1-linie`

Afişarea elementelor de pe diagonala secundară se poate realiza astfel:

```
for (i=0; i<n; i++) cout <<a[i][n-i-1]<<' ';
```

Parcurgerea elementelor de sub diagonala principală

Să presupunem că dorim să calculăm suma elementelor de sub diagonala principală (exclusiv). Observăm că, pe linia `i` (`i` variind de la `1` la `n-1`), indicele de coloană variază între `0` şi `i-1` (deoarece pe linia `i` elementul `a[i][i]` este deja pe diagonala principală).

```
for (s=0, i=1; i<n; i++)
    for (j=0; j<i; j++) s+=a[i][j];
```

Exerciţii

1. Modificaţi secvenţa de instrucţiuni precedentă astfel încât să calculeze suma elementelor de deasupra diagonalei principale.
2. Modificaţi secvenţa de instrucţiuni precedentă astfel încât să calculeze suma elementelor de sub diagonala secundară.
3. Modificaţi secvenţa de instrucţiuni precedentă astfel încât să calculeze suma elementelor de deasupra diagonalei secundare.

4.5. *Şiruri de caractere*

Un şir de caractere este o succesiune de caractere terminată cu caracterul `NULL`. Caracterul `NULL` este caracterul care are codul ASCII `0`.

În limbajul C/C++ nu există un tip special pentru şirurile de caractere, acestea pot fi implementate ca vectori de caractere. Prin urmare, declararea unei variabile `nume`, ce reprezintă un şir de `Lg` caractere, se poate realiza astfel:

```
char nume[Lg];
```

Putem iniţializa o variabilă şir de caractere chiar de la declarare, cu o constantă şir (în acest caz, lungimea şirului `Lg` poate lipsi, fiind determinată automat).

Exemplu

```
char s1[]="Mama are mere", s2[50]="Vasile nu";
```

Pentru şirul `s1`, lungimea a fost determinată automat: `13` octeţi (câte unul pentru fiecare dintre cele 13 caractere din şir) + `1` octet suplimentar pentru marcajul de sfârşit de şir, deci în total `14` octeţi. Pentru `s2`, lungimea a fost specificată (`50` de octeţi), primii `9` octeţi fiind iniţializaţi cu cele `9` caractere ale şirului `"Vasile nu"`, iar cel de-al zecelea octet fiind iniţializat cu marcajul de sfârşit al şirului.

Şirurile de caractere pot fi prelucrate la nivel de caracter (pot fi parcurse caracter cu caracter, ca un vector de caractere) sau pot fi prelucrate la nivel de structură (cu ajutorul funcţiilor existente în bibliotecile limbajului).

Observaţie

Din modul de reprezentare a unui şir de caractere deducem că o constantă caracter (de exemplu, `'a'`) nu este echivalentă cu o constantă şir de caractere (de exemplu cu `"a"`). Constanta `'a'` este stocată pe un singur octet care conţine codul ASCII al caracterului. Constanta `"a"` este stocată pe doi octeţi (primul conţine codul ASCII al caracterului, iar al doilea, marcajul de sfârşit de şir – `NULL`).

4.6. *Citirea şi afişarea şirurilor de caractere*

Să presupunem că am declarat un şir `s` de maxim `100` de caractere astfel:

```
char s[100];
```

Citirea unui şir de caractere în limbajul C++

Citirea şirului `s` se poate face utilizând operatorul uzual de citire >>:

```
cin >> s;
```

În acest caz, se vor citi în şirul `s` toate caracterele până la primul spaţiu. De exemplu, dacă fluxul de intrare conţine caracterele `Mama are mere`, după citire, şirul `s` va fi `Mama`.

Atenţie! Dacă numărul de caractere până la primul spaţiu este mai mare decât lungimea declarată a şirului, citirea se face în continuare şi puteţi distruge astfel date sau cod de program. Efectul este indefinit, dar cu siguranţă este *eroare*!

Acest mod de funcţionare a operatorului >> poate fi inadecvat în multe situaţii. Din acest motiv, în clasa `istream` (pe care o utilizăm când realizăm citiri de la tastatură) există două funcţii speciale pentru citirea caracterelor şi a şirurilor de caractere: `get()` şi `getline()`. Aceste două funcţii sunt incluse în clasa `istream` (se numesc funcţii membre) şi pentru a le putea utiliza trebuie să specificăm fluxul de intrare din care citim şi numele funcţiei, separate prin *operatorul de selecţie* `'.'`: `cin.get()` sau `cin.getline()`.

Funcţia `getline()` are următorul format:

```
getline (char * s, int n, char c='\n');
```

Primul parametru este întotdeauna şirul în care se citeşte. Al doilea parametru reprezintă numărul maxim de caractere care se pot citi. Al treilea parametru, dacă apare, este un caracter delimitator (dacă nu apare, acest caracter este considerat implicit ca fiind '\n'). Ca efect, funcţia citeşte caractere de la intrare până la întâlnirea caracterului delimitator sau până ce a citit exact n-1 caractere.

De exemplu, dacă dorim să citim în şirul s o linie de la tastatură, apelăm funcţia getline() cu doi parametri (deoarece marcajul de sfârşit de linie este '\n'):

```
cin.getline(s,100);
```

Dacă dorim să citim de la tastatură toate caracterele până la întâlnirea caracterului '.', vom apela funcţia getline(), cu trei parametri, astfel:

```
cin.getline(s, 100, '.');
```

Funcţia get() are mai multe forme de apel. Dacă dorim să o utilizăm pentru citirea şirurilor de caractere, apelul este similar cu apelul funcţiei getline(). Şi efectul este asemănător. Diferenţa constă în faptul că getline() extrage din fluxul de intrare caracterul delimitator, în timp ce get() nu îl extrage.

Putem apela funcţia get() fără nici un parametru (caz în care returnează un caracter citit de la intrare) sau cu un parametru de tip char (caz în care parametrul va avea, după citire, valoarea caracterului citit de la intrare).

Observaţie

La citire, marcajul de sfârşit de şir NULL este automat plasat la sfârşitul şirului.

Citirea unui şir de caractere în limbajul C

Un şir de caractere poate fi citit cu ajutorul funcţiei de citire scanf(), utilizând specificatorul de format %s. De exemplu:

```
char s[100];
scanf("%s", s);
```

Observaţi că nu am transmis funcţiei scanf() adresa şirului s (&s). Acest lucru se datorează faptului că numele oricărui vector (deci, implicit, şi al unui şir de caractere) este adresa primului element din vector.

Vor fi citite caracterele care încep cu primul caracter diferit de un caracter alb (spaţiu, tab sau *newline*) până la întâlnirea unui caracter alb. Dacă numărul de caractere citite este mai mare decât lungimea declarată a şirului, veţi obţine eroare[24] la execuţia programului.

24. Eroarea depinde de sistemul de operare şi de mediul de programare uitilizat. De exemplu, dacă lucraţi sub sistemul de operare Linux, veţi obţine mesajul de eroare Segmentation Fault. Acest mesaj de eroare se obţine ori de câte ori încercaţi să accesaţi o zonă de memorie nealocată. Dacă utilizaţi mediul de programare Borland, comportamentul programului este imprevizibil, dar cu maximă probabilitate aplicaţia se va bloca.

Dacă doriți să limitați numărul de caractere care se citesc, puteți indica în specificatorul de format numărul maxim de caractere. De exemplu:

```
scanf ("%9s", s);
```

Observați că, pentru a citi corect un șir de 10 caractere, în specificatorul de format am precizat lungimea maximă 9, deoarece este necesar un octet suplimentar pentru a memora marcajul de sfârșit de șir (caracterul NULL).

Funcția `scanf()` nu permite citirea șirurilor de caractere care conțin caractere albe. Pentru a citi șiruri de caractere care conțin spații sau caractere tab se poate utiliza funcția `gets()`, declarată în fișierul antet `stdio.h`. Prototipul funcției `gets()` este:

```
char *gets(char *s);
```

Funcția citește în șirul s caracterele introduse de la tastatură, până la întâlnirea caracterului marcaj de sfârșit de linie (*newline*). Caracterul *newline* este citit și convertit automat în caracterul NULL.

Observați că, în prototipul funcției `gets()`, faptul că funcția are ca parametru un șir de caractere și returnează ca rezultat un șir de caractere este specificat într-o formă neobișnuită. Practic, se indică drept parametru/rezultat al funcției un pointer cu tipul de bază `char` (adresa de memorie a unui caracter). În secțiunea următoare vom studia tipul pointer, precum și legătura dintre pointeri și tablouri. De exemplu, pentru a citi un șir s care, eventual, poate conține și caractere tab sau spații apelăm funcția `gets()` astfel:

```
gets(s);
```

Afișarea unui șir de caractere în limbajul C++

Afișarea unui șir de caractere se face cu ajutorul operatorului de scriere <<:

```
cout << s;
```

Se vor scrie caracterele până la întâlnirea primului marcaj de sfârșit de șir (NULL). De exemplu, dacă s este `"mama are \n mere"` se va afișa:

```
mama are
 mere
```

Dacă s este `"mama are \n mere \0 de aici nu mai scrie"` efectul va fi același (scrie până la `'\0'`, caracterul NULL).

Afișarea unui șir de caractere în limbajul C

Afișarea unui șir de caractere se poate face cu ajutorul funcției `printf()`, utilizând specificatorul de format `%s`. De exemplu:

```
char s[100];
printf("%s", s);
```

În fișierul antet `stdio.h` este declarată o funcție specială pentru afișarea șirurilor de caractere, funcția `puts()`. Prototipul acestei funcții este:

```
int puts(const char *s);
```

Funcția afișează caracterele șirului `s` până la întâlnirea caracterului `NULL` și apoi afișează un caracter *newline* (de trecere la linie nouă). În caz de afișare cu succes, funcția returnează o valoare pozitivă; în caz contrar, returnează valoarea `-1`. Prin urmare, apelul:

```
puts(s);
```

este echivalent cu:

```
printf("%s\n", s);
```

4.7. *Tipul pointer*

Un *pointer* este o dată care are ca valori adrese de memorie.

Facilitățile oferite de limbajul C/C++ pentru lucrul cu pointeri reprezintă unul dintre cele mai importante atuuri ale limbajului. Pointerii sunt utilizați cu precădere în următoarele situații: lucrul cu tablouri, transferul parametrilor funcțiilor, accesul direct la memorie și alocarea dinamică a memoriei. Pentru a înțelege mai bine modul de lucru cu tablouri și cu șiruri de caractere, ne interesează pointerii de date (pointeri care conțin adresa zonei de memorie a unei variabile).

Declararea unui pointer de date

Declararea unui pointer de date are formatul general:

```
tip * variabila_pointer;
```

unde `tip` reprezintă *tipul de bază* al pointerului, care indică tipul datelor memorate la adresa conținută în `variabila_pointer`.

Exemple

```
int * p1;
```

`p1` este un pointer cu tipul de bază `int`, deci va conține adresa unei zone de memorie la care este memorat un număr întreg (de tip `int`).

```
char * s;
```

`s` este un pointer cu tipul de bază `char`, deci va conține adresa unei zone de memorie la care este memorat un caracter.

Observații

1. Declarația precedentă este asemănătoare cu declarația unei variabile uzuale, faptul că variabila declarată este un pointer fiind indicat de prezența asteriscului.
2. Tipul de bază al pointerului poate fi orice tip al limbajului, inclusiv tipul `void`. Un pointer cu tipul de bază `void` (`void *`) se numește pointer generic.
3. Indicarea tipului de bază al pointerului este esențială. Adresa unei zone de memorie este de fapt adresa primului octet din zona respectivă (o valoare numerică). Indicând tipul de bază al pointerului, se poate determina, în primul rând, dimensiunea zonei de memorie a cărei adresă este memorată în pointer (de exemplu, 1

octet pentru `char *`, 4 octeți pentru `long int *` etc). Dimensiunea zonei de memorie nu este suficientă, pentru a afla valoarea memorată într-o anumită zonă de memorie trebuie să știm și codul utilizat pentru reprezentarea datelor în acea zonă. De exemplu, într-o zonă de 4 octeți s-ar putea afla un număr natural de tip `unsigned long int`, un număr întreg de tip `long int`, un număr real de tip `float` sau un șir de 4 caractere. Cum decodificăm o secvență de octeți dacă nu știm codul utilizat pentru reprezentarea datelor în acea secvență? Prin urmare, tipul de bază al pointerului ne indică atât dimensiunea zonei a cărei adresă de început este memorată în pointer, cât și codul utilizat pentru reprezentarea datelor în acea zonă.

4. Există o constantă pointer specială, denumită `NULL`. Semnificația acestei constante este „pointerul nu conține adresa nici unei zone de memorie". Valoarea acestei constante este `0`.

Operații cu pointeri

1. Operația de referențiere

Este operația prin care putem obține adresa de memorie a unei variabile. Rezultatul acestei operații este un pointer (care are ca valoare adresa primului octet al zonei de memorie în care este stocată variabila). Referențierea unei variabile se realizează cu ajutorul operatorului unar `&` (numit operatorul de referențiere sau operatorul adresă):

```
&variabila
```

2. Operația de dereferențiere

Este operația prin care putem accesa valoarea memorată într-o zonă de memorie a cărei adresă de început este memorată într-un pointer. Dereferențierea unui pointer se realizează utilizând operatorul unar * (numit operator de dereferențiere):

```
*variabila_pointer
```

Această expresie are ca valoare conținutul zonei de memorie a cărei adresă este memorată în `variabila_pointer`.

Exemplu

```
int i=10, *p;
p=&i;
```

Referențiere – atribuim pointerului `p` adresa variabilei `i`.

```
cout << *p;
```

Dereferențiere – afișez conținutul zonei de memorie a cărei adresă este memorată în `p` (valoarea `10`).

3. Incrementare/decrementare

Asupra unui pointer se poate aplica operatorul de incrementare `'++'` și operatorul de decrementare `'--'`. Ca efect, se mărește/micșorează adresa memorată în pointer

cu numărul de octeți necesari pentru a memora o dată de tipul de bază al pointerului (`sizeof(tip)`).

Exemple

```
long int * p;
p++;
```

Adresa memorată în `p` se mărește cu `4=sizeof(long int)`.

```
char * p;
p--;
```

Adresa memorată în `p` se micșorează cu `1=sizeof(char)`.

Efectul este natural: pointerul indică următorul (++) sau precedentul (--) element de tipul său de bază.

4. Adunarea/scăderea dintre un pointer și un număr întreg

Se poate realiza adunarea (+) dintre un pointer și un întreg sau scăderea dintre un pointer și un întreg: `p+n` sau `p-n`. Ca efect, adresa memorată în pointer se mărește (+) sau se micșorează (-) cu `n*sizeof(tip)`.

Deci, `p+n` ar indica al `n`-lea element de tipul de bază al pointerului care urmează după adresa `p`; `p-n` ar indica al `n`-lea element de tipul de bază al pointerului care precedă elementul de la adresa `p`.

5. Scăderea dintre doi pointeri

Se poate realiza scăderea între doi pointeri numai dacă aceștia au același tip de bază. Rezultatul este o valoare întreagă care reprezintă diferența dintre adrese, raportată la `sizeof(tip)`.

6. Comparații

Asupra pointerilor care au același tip de bază se pot aplica operatorii relaționali și de egalitate, cu semnificația uzuală.

Exemplu

```
if (p==NULL) ...
```

7. Afișarea unui pointer

Pentru a afișa valoarea unui pointer în limbajul C se utilizează funcția `printf()` cu specificatorul de format `%p`. De exemplu:

```
int a, *q=&a;
printf("%p", q);
```

Pe ecran va fi afișată adresa variabilei `a`, în baza 16.

Afișarea unui pointer în limbajul C++ se face cu ajutorul operatorului uzual de afișare. De exemplu:

```
cout<<q;
```

4.8. *Legătura dintre pointeri şi tablouri*

Numele unui tablou este un *pointer constant* care are ca valoare adresa primului element din tablou.

Să considerăm următoarea declaraţie:

```
tip v[LgMax];
```

Următoarele expresii sunt echivalente:

`v`	`&v[0]`	adresa primului element din tablou
`v+i`	`&v[i]`	adresa elementului de pe poziţia `i` din tablou
`*v`	`v[0]`	primul element din tablou
`*(v+i)`	`v[i]`	elementul de pe poziţia `i` din tablou
`v++`	`v[1]`	elementul de pe poziţia `1`

Operaţiile cu pointeri reprezintă o alternativă pentru adresarea elementelor unui tablou.

Exemplu

Să considerăm `s` un şir de maxim 100 de caractere.

```
char s[100];
```

Să determinăm lungimea efectivă a şirului de caractere `s`. Determinarea lungimii unui şir se poate face parcurgând şirul până la întâlnirea marcajului de sfârşit de şir şi numărând caracterele parcurse.

Pentru a parcurge şirul putem utiliza variabila întreagă `lungime` (care indică poziţia curentă în şir, iar la sfârşit lungimea şirului):

```
int lungime;
for (lungime=0; s[lungime]; lungime++);
```

Şirul poate fi parcurs şi cu ajutorul unui pointer `p` care iniţial va avea ca valoare adresa de început a şirului. Pointerul va fi incrementat cât timp valoarea lui nu este adresa zonei de memorie care conţine caracterul `NULL`. Lungimea şirului este diferenţa dintre valoarea pointerului `p` (care indică la sfârşit adresa caracterului `NULL`) şi valoarea pointerului `s` (care indică adresa primului caracter din şir).

```
char *p;
for (p=s; *p; p++);
lungime=p-s;
```

Observaţii

1. Acum putem explica de ce atribuirea `a=b` produce mesajul de eroare *„Lvalue required"* atunci când încercăm să atribuim unui tablou `a` tabloul `b`: membrul stâng al acestei atribuiri este un pointer constant. Această atribuire este tentativa de a modifica un pointer constant (încercăm să atribuim pointerului constant `a` valoarea pointerului `b`).

2. O matrice este un vector de vectori, deci numele său este un pointer la prima linie din matrice. Să considerăm următoarea declarație:

```
tip v[LgMax][LgMax];
```

Următoarele expresii sunt echivalente:

`v`	`&v[0]`		adresa primei linii din matrice
`v+i`	`&v[i]`		adresa liniei `i` din matrice
`*v`	`v[0]`	`&v[0][0]`	adresa primului element de pe linia 0
`*(v+i)`	`v[i]`	`&v[i][0]`	adresa primului element de pe linia `i`
`*(*(v+i)+j)`		`v[i][j]`	elementul de pe linia `i` și coloana `j`

Utilizarea funcțiilor standard de lucru cu șiruri de caractere

În fișierul antet `string.h` sunt declarate numeroase funcții de lucru cu șiruri de caractere.

Determinarea lungimii efective a unui șir de caractere

În secțiunea precedentă am determinat lungimea efectivă a unui șir de caractere parcurgând șirul până la întâlnirea caracterului `NULL`. O variantă mai simplă este de a apela funcția `strlen()`. Prototipul acestei funcții este:

```
unsigned strlen(const char *s);
```

Apelul `strlen(s)` returnează lungimea șirului `s`.

Copierea unui șir de caractere într-un alt șir de caractere

Copierea unui șir de caractere (denumit `sursa`) într-un alt șir de caractere (denumit `destinatia`) se poate realiza cu ajutorul funcției `strcpy()`. Prototipul funcției `strcpy()` este:

```
char * strcpy(char *destinatia, const char *sursa);
```

Apelul `strcpy(destinatie, sursa)` copiază șirul `sursa`, caracter cu caracter, în șirul `destinatie`, până după copierea caracterului `NULL` și returnează adresa de început a șirului `destinatie`.

Efectul acestui apel este echivalent cu următoarea secvență de instrucțiuni:

```
int i;
for (i=0; sursa[i]; i++) //parcurg sirul sursa
     //copiez caracterul curent in sirul destinatie
     destinatie[i]=sursa[i];
destinatie[i]=NULL; //plasez la sfarsit caracterul NULL
```

Exercițiu

Scrieți o secvență echivalentă care să utilizeze pointeri pentru parcurgerea șirurilor.

Observație

Spațiul de memorie alocat șirului `destinatie` trebuie să fie suficient de mare pentru a memora caracterele șirului `sursa`.

Copierea unui prefix al unui șir de caractere

Pentru a copia primele `n` caractere din șirul de caractere `sursa` în șirul de caractere `destinatie` se poate utiliza funcția `strncpy()`. Prototipul funcției `strncpy()` este:

```
char * strncpy(char *destinatie, char *sursa, unsigned n);
```

Apelul `strncpy(destinatie,sursa,n)` are ca efect copierea primelor maxim `n` caractere din șirul `sursa` în șirul `destinatie` și returnează adresa de început a șirului `destinatie`. Dacă lungimea șirului `sursa` este mai mică decât `n`, se copiază întreg șirul `sursa` în șirul `destinatie` și la sfârșitul șirului `destinatie` se plasează caracterul `NULL`. Dacă lungimea șirului `sursa` este mai mare sau egală cu `n`, atunci la sfârșitul șirului `destinatie` nu va fi plasat caracterul `NULL`.

Efectul acestui apel este echivalent cu următoarea secvență de instrucțiuni:

```
int i=-1;
do
  {i++; destinatie[i]=sursa[i];}
while (sursa[i] && i<n-1);
```

Concatenarea a două șiruri de caractere

Pentru a concatena șirul de caractere `destinatie` cu șirul de caractere `sursa` se utilizează funcția `strcat()`. Prototipul funcției `strcat()` este:

```
char * strcat(char *destinatie, const char *sursa);
```

Apelul `strcat(destinatie,sursa)` copiază caracterele șirului `sursa` la sfârșitul șirului `destinatie` și returnează adresa de început a șirului obținut după concatenare (`destinatie`).

Efectul acestui apel este echivalent cu următoarea secvență de instrucțiuni:

```
int i, lgd=strlen(destinatie);
for (i=0; sursa[i]; i++)
     destinatie[i+lgd]=sursa[i];
destinatie[lgd+i]=NULL;
```

Observație

Zona de memorie alocată șirului `destinatie` trebuie să fie suficient de mare pentru a reține rezultatul concatenării.

Concatenarea unui șir de caractere cu un prefix al altui șir de caractere

Pentru a concatena primele `n` caractere din șirul de caractere `sursa` la șirul de caractere `destinatie` se poate utiliza funcția `strncat()`. Prototipul funcției

```
char *strncat(char *destinatie, char *sursa, unsigned n);
```

Apelul `strncat(destinatie,sursa,n)` are ca efect copierea primelor maxim `n` caractere din şirul `sursa` la sfârşitul şirului `destinatie` şi returnează adresa de început a şirului `destinatie`. Dacă lungimea şirului `sursa` este mai mică decât `n`, se concatenează întreg şirul `sursa` la şirul `destinatie`. La sfârşitul şirului obţinut în urma concatenării se plasează caracterul `NULL`.

Efectul acestui apel este echivalent cu următoarea secvenţă de instrucţiuni:

```
int i, lgd=strlen(destinatie);
for (i=0; sursa[i] && i<n; i++)
    destinatie[i+lgd]=sursa[i];
destinatie[i+lgd]=NULL;
```

Compararea a două şiruri de caractere

Pentru a compara două şiruri de caractere se poate utiliza funcţia `strcmp()`. Prototipul acestei funcţii este:

```
int strcmp(const char *s1, const char *s2);
```

Apelul `strcmp(s1,s2)` compară din punct de vedere lexicografic şirul de caractere `s1` cu şirul de caractere `s2` şi returnează o valoare:

`< 0`, dacă `s1<s2` (în ordine lexicografică, ordinea cuvintelor din dicţionar);
`==0`, dacă `s1==s2` (`s1` şi `s2` sunt egale);
`> 0`, dacă `s1>s2` (în ordine lexicografică).

Observaţie

Spunem că şirul `s1` precedă din punct de vedere lexicografic şirul `s2` dacă există un număr natural $n \leq$ `strlen(s1)`, astfel încât `s1[i]=s2[i]` pentru orice $0 \leq$ `i<n` şi `s1[n]<s2[n]`.

Efectul acestui apel este echivalent cu următoarea secvenţă de instrucţiuni:

```
int ok, i; //variabila ok va retine rezultatul testului
for (i=0; s1[i] && s2[i] && s1[i]==s2[i]; i++);
if (!s1[i] && !s2[i]) ok=0;
   else
   if (s1[i]<s2[i]) ok=-1;
      else ok=1;
```

Observaţie

Utilizând funcţia `strcmp()`, la comparare se face distincţie între literele mari şi literele mici. Dacă dorim o comparare care să nu facă distincţie între majuscule şi minuscule (*case insensitive*), putem utiliza funcţia `stricmp()`:

```
int stricmp(const char *s1, const char *s2);
```

Căutarea primei apariții a unui caracter într-un șir

Pentru a căuta prima apariție a unui caracter într-un șir se poate utiliza funcția `strchr()`. Prototipul acestei funcții este:

```
char * strchr(const char *s, int c);
```

Apelul `strchr(s,c)` caută prima apariție a caracterului `c` în șirul `s` și returnează un pointer care are ca valoare adresa primului caracter din șir egal cu `c` (dacă un astfel de caracter există) sau `NULL` (dacă în `s` caracterul `c` nu apare).

Efectul acestui apel este echivalent cu următoarea secvență de instrucțiuni:

```
char *p;
for (p=s; *p && *p!=c; p++);
if (!*p) p=NULL;
```

Exercițiu

Rescrieți această secvență de instrucțiuni utilizând indici pentru parcurgerea șirului `s`.

Căutarea primei apariții a unui șir într-un alt șir

Pentru a căuta prima apariție a unui șir într-un alt șir se poate utiliza funcția `strstr()`. Prototipul acestei funcții este:

```
char *strstr(const char *s1, const char *s2);
```

Apelul `strstr(s1,s2)` determină prima apariție a șirului `s2` în șirul `s1` și returnează un pointer către începutul primei apariții a lui `s2` în `s1` sau `NULL` (dacă `s2` nu apare în `s1`).

Efectul acestui apel este echivalent cu următoarea secvență de instrucțiuni:

```
int i, j, lg1=strlen(s1), lg2=strlen(s2);
char *p=NULL;
for (i=0; i<=lg1-lg2; i++)
    {
     for (j=0; s2[j] && s2[j]==s1[i+j]; j++);
     if (!s2[j]) {p=s1+i; break;}
    }
```

Transformări între litere mici și litere mari

Pentru a transforma toate literele mari dintr-un șir în litere mici se poate utiliza funcția `strlwr()`. Prototipul acestei funcții este:

```
char *strlwr(char *s);
```

Pentru a transforma toate literele mici dintr-un șir în litere mari se poate utiliza funcția `strupr()`. Prototipul acestei funcții este:

```
char *strupr(char *s);
```

Exercițiu

Scrieți câte o instrucțiune care să aibă un efect echivalent cu funcția `strlwr()`, respectiv cu `strupr()`.

Separarea unităților lexicale dintr-un șir de caractere

Numim unitate lexicală o secvență de caractere din șir delimitată de separatori. Separatorii depind de problema concretă și sunt memorați într-un șir de caractere (pe care l-am denumit `separatori`).

Funcția `strtok()` permite extragerea succesivă a unităților lexicale dintr-un șir de caractere. Prototipul acestei funcții este:

```
char *strtok(char *s, const char *separatori);
```

Referitor la modul de apel, această funcție este mai specială. Prima dată vom apela funcția sub forma:

```
strtok(s,separatori);
```

După primul apel, funcția va returna adresa de început a primei unități lexicale din șirul `s`. După acest prim apel, șirul `s` este modificat (primul separator de după prima unitate lexicală este înlocuit de caracterul `NULL`. Dacă șirul `s` este vid sau conține numai separatori, atunci funcția returnează `NULL`.

Pentru a extrage succesiv și celelalte unități lexicale din șirul `s` putem apela din nou funcția `strtok()`, dar de data aceasta primul parametru trebuie să fie `NULL` (altfel, funcția returnează tot prima unitate lexicală din `s`):

```
strtok(NULL, separatori);
```

Acest apel returnează adresa de început a următoarei unități lexicale din șirul `s` (sau `NULL`, dacă în șirul `s` nu mai există unități lexicale). După fiecare apel, primul separator de după unitatea lexicală identificată este înlocuit de caracterul `NULL`.

Exemplu

Să considerăm următoarele șiruri:

```
char s[]=" Vai, vai!   Ce necaz!!!!\nUnde e pisica?";
char sep[]=" ,!?\n;.:";                    //sirul de separatori
```

La primul apel:

```
char *p=strtok(s,sep);
```

variabilei `p` i se atribuie adresa de început a primei unități lexicale (`Vai`), iar caracterul separator `' '` de pe poziția 4 este înlocuit de caracterul `NULL`. Dacă vom afișa șirul a cărui adresă de început este memorată în `p`:

```
cout<<p;
```

vom obține pe ecran prima unitate lexicală:

```
Vai
```

La al doilea apel:

```
p=strtok(NULL, sep);
```

variabilei p i se atribuie adresa de început a celei de-a doua unități lexicale (vai), iar caracterul separator '!' este înlocuit de NULL.

La al treilea apel:

```
p=strtok(NULL, sep);
```

variabilei p i se atribuie adresa de început a celei de-a treia unități lexicale (Ce), iar caracterul separator ' ' este înlocuit de NULL.

Pentru a extrage toate unitățile lexicale din șirul s separate de caractere din șirul de separatori sep putem utiliza următoarea secvență de instrucțiuni:

```
int i=0;
p=strtok(s, sep);
while (p)
      {i++;
       cout<<"Unitatea lexicala nr. "<<i<<" este ";
       cout<<p<<endl;
       p=strtok(NULL, sep);}
```

În urma execuției acestei secvențe de instrucțiuni pentru șirurile declarate anterior, vom obține pe ecran:

```
Unitatea lexicala nr. 1 este Vai
Unitatea lexicala nr. 2 este vai
Unitatea lexicala nr. 3 este Ce
Unitatea lexicala nr. 4 este necaz
Unitatea lexicala nr. 5 este Unde
Unitatea lexicala nr. 6 este e
Unitatea lexicala nr. 7 este pisica
```

Observație

De la un apel la altul, șirul de separatori se poate modifica. De exemplu, să considerăm că în șirul s este memorată o relație de forma:

```
<rezultat> = <operand_1> <operator> <operand_2>
```

unde operatorul poate fi '+', '-', '*' sau '/'.

Pentru a extrage cele trei unități lexicale (rezultat, operand_1 și operand_2) putem proceda astfel:

```
p=strtok(s,"=");
cout<<"Rezultat:"<<p<<endl;
p=strtok(NULL,"+-*/ ");
cout<<"Operand 1:"<<p<<endl;
p=strtok(NULL,"+-*/ ");
cout<<"Operand 2:"<<p<<endl;
```

Funcții de conversie

În fișierul antet `stdlib.h` sunt declarate funcții care permit conversia unui număr întreg într-un șir de caractere:

```
char *itoa(int nr, char *sir, int baza);
char *ltoa(long nr, char *sir, int baza);
char *ultoa(unsigned long nr, char *sir, int baza);
```

Funcția `itoa()` convertește un număr de tip `int`, funcția `ltoa()` convertește un număr de tip `long`, iar funcția `ultoa()` convertește un număr de tip `unsigned long`. Șirul de caractere `sir` va conține reprezentarea numărului `nr` în baza de numerație `baza` (2≤baza≤36).

Exercițiu

În fișierul antet `stdlib.h` sunt declarate funcții care realizează și conversia în sens invers (din șir de caractere în valoare numerică) – `atoi()`, `atol()`, `atof()`, `strtod()`, `_strtold()`, `strtol()`, `strtoul()`. Studiați aceste funcții utilizând sistemul de autodocumentare (*Help*).

4.9. *Tipul de date* `struct`

Tablourile sunt structuri de date care conțin date de *același tip*. Totuși, apare frecvent necesitatea de a utiliza structuri de date care să conțină date *de tipuri diferite*, reunite sub un nume comun. O astfel de structură este cunoscută sub denumirea de *articol* (sau *înregistrare* sau *structură*) și este implementată în limbajul C/C++ cu ajutorul tipului de date `struct`. Datele înglobate într-un articol sunt denumite *câmpuri* sau *membri*.

Deoarece câmpurile nu mai sunt obligatoriu de același tip, ele nu mai pot fi referite prin intermediul indicilor. Din acest motiv, fiecare câmp are un nume distinct, care este utilizat pentru a referi câmpul respectiv.

Declararea unui tip `struct`

Declararea unui tip `struct` are următoarea sintaxă:

```
struct [nume_tip]
{lista_declaratii_campuri} [lista_variabile];
```

Efect

Am declarat un tip `struct` cu numele `nume_tip` care conține câmpurile declarate în `lista_declaratii_campuri` și o listă de variabile având tipul `struct` definit.

Observații

1. Lista de variabile este opțională. Nu este obligatoriu ca atunci când definim tipul să declarăm și variabile de tipul respectiv. Acest lucru se poate realiza și ulterior, astfel:

```
struct nume_tip lista_variabile;
```
sau (valabil numai în C++, nu și în C):
```
nume_tip lista_variabile;
```
2. Numele tipului `struct` este opțional, deci puteți declara și tipuri anonime. Dar în acest caz este obligatoriu ca lista de variabile să apară (tipul fiind anonim, nu îl mai putem utiliza în declarații de variabile ulterioare).

O declarație de câmpuri are sintaxa unei declarații de variabile uzuale:

```
tip nume_camp1, nume_camp2, ..., nume_campk;
```

Exemple

1. Să declarăm un tip `struct` denumit `Punct` care să reprezinte un punct în planul cartezian:

```
struct Punct
       {double x, y;};
```
Un articol de tip `Punct` conține două câmpuri de tip `double`: `x` (abscisa punctului) și `y` (ordonata punctului). În acest caz, câmpurile sunt de același tip. Nu am declarat și variabile de tip `Punct`. Acest lucru se poate realiza ulterior. De exemplu, declarăm `P1`, și `P2` două variabile de tip `Punct`:

```
struct Punct P1, P2;
```
sau (valabil numai în C++, nu și în C):
```
Punct P1, P2;
```
2. Să declarăm o structură care să conțină informații despre un elev: numele, mediile la cele 14 discipline de învățământ, media generală, adresa și numărul de telefon.

```
struct Elev
      {char nume[30], adr[60];
       float medii[14], mg;
       unsigned long tel; } Clasa[30];
```
Observați că la definirea tipului `Elev` am declarat și o variabilă (`Clasa`) care este un vector cu 30 de componente de tip `Elev`.

3. Să declarăm o structură care să rețină date calendaristice:

```
struct Data
        {int zi, an;
         char luna[15];};
```

Referirea la un câmp al unei structuri

Referirea la un câmp al unui articol se realizează specificând numele structurii și numele câmpului respectiv, separate prin *operatorul de selecție directă* `'.'`:

```
variabila_structura.camp
```

Exemple

1. Pentru a ne referi la abscisa punctului `P1`, vom scrie `P1.x`.
2. Pentru a ne referi la numele primului elev memorat în vectorul `Clasa`, vom scrie `Clasa[0].nume`.

Inițializarea unei structuri

Ca oricare alte variabile, la declararea unei variabile de tip `struct` aceasta poate fi inițializată. La inițializare se precizează, în ordine, valorile ce vor fi atribuite câmpurilor structurii. Valorile sunt separate prin virgulă și încadrate între paranteze acolade. Valorile sunt atribuite în ordine câmpurilor structurii și trebuie să corespundă ca tip cu acestea. Numărul de valori precizate la inițializare poate fi egal cu numărul de câmpuri ale structurii sau mai mic (caz în care numai primele câmpuri vor fi inițializate).

De exemplu, să declarăm o variabilă de tipul `struct Data` definit anterior și să o inițializăm cu 31-12-2004.

```
struct Data azi={31, 2004,"Decembrie"};
```

Citirea și afișarea structurilor

Pentru a citi/afișa o structură, trebuie să citim/afișăm separat fiecare câmp al structurii.

Operații cu structuri

Singura operație care se poate realiza la nivel de structură este atribuirea. Mai exact, unei variabile de tip `struct` i se poate atribui o altă variabilă de același tip `struct`. De exemplu, putem declara o variabilă de tip `struct Data` denumită `ieri` și să o inițializăm cu valoarea variabilei `azi`:

```
struct Data ieri;
ieri=azi;
```

4.10. *Asocierea unui nume pentru un tip de date*

Tipurile de date predefinite ale limbajului (de exemplu, `int`, `char`, `float` etc.) se identifică printr-un nume. În secțiunea precedentă, am văzut și că atunci când definim un tip `struct` îi putem asocia un nume.

Programatorul poate asocia unui tip de date un nume, indiferent dacă acesta este un tip predefinit sau definit de el, utilizând construcția `typedef`:

```
typedef descriere_tip nume_tip;
```

Exemple

1. Să asociem tipului predefinit `float` numele (mai sugestiv pentru noi) `REAL`:

```
typedef float REAL;
```

2. Să asociem tipului `int *` (pointer cu tipul de bază `int`) numele `PInt`:

```
typedef int * PInt;
```

3. Să asociem numele `Polinom` unui tip articol care să conțină gradul polinomului și coeficienții:

```
typedef struct
        {int grad;
         double coef[100];}  Polinom;
```

4. Să asociem un nume tipului tablou cu `100` de componente de tip `int`:

```
typedef int Vector[100];
```

4.11. *Aplicații*

Temperaturi

Se citesc de la tastatură temperaturile înregistrate la Institutul meteorologic în fiecare dintre zilele lunii ianuarie. Scrieți un program care să numere în câte zile ale lunii au fost temperaturi strict pozitive și de câte ori s-a schimbat semnul temperaturilor (s-a înregistrat o trecere de la temperaturi pozitive la temperaturi negative sau invers) în decursul lunii ianuarie.

Soluție

Vom memora temperaturile din luna ianuarie într-un vector `T` cu `31` de elemente. Vom parcurge vectorul de la primul până la ultimul element, testând pentru fiecare element din vector dacă este o temperatură pozitivă. În caz afirmativ, numărăm încă o zi cu temperaturi pozitive (în acest scop utilizăm variabila `nrp`).

Pentru a determina numărul de schimbări de semn din vectorul `T` este suficient ca în timpul parcurgerii să verificăm dacă există în vector două elemente situate pe poziții consecutive și de semne contrare. Schimbările de semn vor fi numărate în variabila `nrs`.

```
#include <iostream.h>
#define NMax 31
int main()
{int T[NMax], nrp, nrs=0, i;
 cout<<"Introduceti temperaturile: ";
 for (i=0; i<31; i++) cin>>T[i];
 nrp=T[0]>0?1:0;
 for (i=1; i<31; i++)
     {if (T[i]>0) nrp++;
      if (T[i-1]*T[i]<0) nrs++;     }
 cout<<"nr de zile cu temperaturi pozitive "<<nrp<<endl;
 cout<<"nr de schimbari de semn "<<nrs<<endl;
 return 0; }
```

Inserare medii

Se citește de la tastatură un număr natural n (1<n≤50) și elementele unui vector cu n componente reale. Să se insereze între oricare două elemente consecutive ale vectorului media lor aritmetică.

Soluție

O primă idee este de a citi elementele vectorului și apoi de a insera între oricare două elemente vecine media lor. Pentru a insera media, trebuie să-i „facem loc", adică trebuie să deplasăm toate elementele care urmează cu o poziție la dreapta. Nu uitați că, prin inserare, numărul de elemente din vector crește!

```
#include <iostream.h>
int main ()
{float a[100];
 int n, i, j;
 cout <<"n="; cin >> n;
 for (i=0; i<n; i++) {cout <<"a["<<i<<"]="; cin>>a[i];}
 for (i=1; i<n; i+=2) //inserez media dintre a[i] si a[i-1]
   {//mut elementele de la i la n-1 cu o pozitie la dreapta
    for (j=n; j>i; j--) a[j]=a[j-1];
    a[i]=(a[i]+a[i-1])/2;        //plasez media pe pozitia i
    n++; }           //maresc numarul de elemente din vector
 for (i=0; i<n; i++) cout <<a[i]<<' ';
 cout<<endl; return 0;}
```

O soluție mai simplă ar fi fost de a plasa în vector elementele citite numai pe poziții de indice par, astfel încât pozițiile de indice impar să rămână disponibile pentru medii, evitând astfel deplasările succesive:

```
#include <iostream.h>
int main ()
{
 float a[100];
 int n, i;
 cout <<"n="; cin >> n;
 for (i=0; i<n; i++)
     {cout <<"a["<<i<<"]=";
      cin>>a[2*i]; }
 for (i=1; i<n; i++)
     a[2*i-1]=(a[2*i]+a[2*i-2])/2;
 for (i=0; i<2*n-1; i++) cout <<a[i]<<' ';
 cout<<endl;  return 0; }
```

Combinare vectori

Fie n un număr natural, n≤20, și A, B doi vectori cu câte n componente întregi. Scrieți un program care să combine cei doi vectori conform relațiilor următoare:

`A[i]=A[i]+B[i]`

`B[i]=A[i]*B[i]`, pentru orice `i=0, 1, 2, ..., n-1`.

Soluție

O primă idee este de a parcurge vectorii simultan și de a actualiza valorile elementelor vectorilor conform relațiilor de mai sus:

```
#include <iostream.h>
int main()
{int n, A[20], B[20], i;
 cout<<"n= "; cin>>n;
 cout<<"Elementele vectorului A: ";
 for (i=0; i<n; i++) cin >> A[i];
 cout<<"Elementele vectorului B: ";
 for (i=0; i<n; i++) cin >> B[i];
 for (i=0; i<n; i++)
     {A[i]=A[i]+B[i];
      B[i]=A[i]*B[i];  }
 cout<<"Vectorul A: ";
 for (i=0; i<n; i++) cout<<A[i]<<' '; cout<<endl;
 cout<<"Vectorul B: ";
 for (i=0; i<n; i++) cout<<B[i]<<' '; cout<<endl;
 return 0; }
```

Executând acest program, observăm că rezultatele sunt *greșite*. Să urmărim execuția acestui program, pas cu pas, pentru `n=2`, `A={1, 2}`, `B={3, 4}`.

i	A	B	Explicație
0	{4, 2}	{12, 4}	Se calculează `A[0]=A[0]+B[0]=1+3=4`. Apoi se calculează `B[0]=A[0]*B[0]=4*3=12`. Observăm că valoarea lui `A[0]` a fost calculată corect, dar valoarea lui `B[0]` este greșită, pentru că în calcul se utilizează noua valoare a lui `A[0]`.
1	{4, 6}	{12, 24}	Se calculează `A[1]=A[1]+B[1]=2+6=6`. Apoi se calculează `B[1]=A[1]*B[1]=6*4=24`. Din nou, valoarea lui `A[1]` a fost calculată corect, dar valoarea lui `B[1]` este greșită.

Pentru a calcula corect și valorile elementelor din vectorul `B`, salvăm valoarea inițială a elementului `A[i]` într-o variabilă auxiliară.

Programul corect este:

```
#include <iostream.h>
int main()
{int n, A[20], B[20], i, aux;
 cout<<"n= "; cin>>n;
```

```
cout<<"Elementele vectorului A: ";
for (i=0; i<n; i++) cin >> A[i];
cout<<"Elementele vectorului B: ";
for (i=0; i<n; i++) cin >> B[i];
for (i=0; i<n; i++)
    {aux=A[i];
     A[i]=A[i]+B[i];
     B[i]=aux*B[i];    }
cout<<"Vectorul A: ";
for (i=0; i<n; i++) cout<<A[i]<<' '; cout<<endl;
cout<<"Vectorul B: ";
for (i=0; i<n; i++) cout<<B[i]<<' '; cout<<endl;
return 0; }
```

Ciurul lui Eratostene

Fie n un număr natural (n≤10000). Să se genereze toate numerele prime mai mici decât n.

Soluție

O primă idee ar fi să parcurgem toate numerele naturale din intervalul [2, n], pentru fiecare număr să verificăm dacă este prim și să afișăm numerele prime determinate.

O idee mai eficientă provine din Antichitate și poartă numele de ciurul lui Eratostene[25]. Ideea este de a „pune în ciur" toate numerele mai mici decât n, apoi de a „cerne" aceste numere până rămân în ciur numai numerele prime. Mai întâi „cernem" (eliminăm din ciur) toți multiplii lui 2, apoi cernem multiplii lui 3 ș.a.m.d.

Vom reprezenta ciurul ca un vector cu 10000 de componente care pot fi 0 sau 1, cu semnificația ciur[i]=1, dacă numărul i este în ciur, și 0, altfel.

Vom parcurge vectorul ciur de la 2 (cel mai mic număr prim), până la $\sqrt{n}$.

Dacă ciur[i] este 1, deducem că i este prim, dar toți multiplii lui nu vor fi (deci eliminăm din ciur toți multiplii lui i).

În final, în vectorul ciur vor avea valoarea 1 doar componentele de pe poziții numere prime.

```
#include <iostream.h>
#define NMax 10000
int main()
{int ciur[NMax], n, i, j;
```

25. Eratostene (276-196 î.e.n.) a fost un important matematician și filosof al Antichității. Deși puține dintre lucrările lui s-au păstrat, a rămas celebru prin metoda rapidă de determinare a numerelor prime și prin faptul că a fost primul care a estimat cu acuratețe diametrul Pământului.

```
cout<<"n= "; cin>>n;
//initial toate numerele sunt in ciur
for (i=2; i<n; i++) ciur[i]=1;
for (i=2; i*i<=n; i++)
    if (ciur[i])//i este prim
       //elimin toti multiplii lui i
       for (j=2; j*i<n; j++)
           ciur[i*j]=0;
for (i=2; i<n; i++)
    if (ciur[i]) cout<<i<<' ';
return 0; }
```

Copii

La ora de educație fizică, profesorul a cerut elevilor să se alinieze. Fiind un profesor pasionat de informatică, a observat imediat că există situații în care un copil de înălțime maximă este așezat lângă un copil de înălțime minimă.

Scrieți un program care să determine câte perechi de elevi alăturați îndeplinesc această condiție.

Soluție

Vom reține înălțimile copiilor, în ordinea în care s-au aliniat aceștia, în vectorul `h`. Determinăm mai întâi înălțimea celui mai înalt copil și înălțimea celui mai scund copil. Parcurgem apoi vectorul, identificând toate perechile de copii alăturați în care primul copil are înălțimea maximă și al doilea, înălțimea minimă sau invers.

```
#include <iostream.h>
#define NMax 40
int main()
{
 int h[NMax], n, i, nr, min, max;
 //citire
 cout<<"n= "; cin>>n;
 for (i=0; i<n; i++) cin>>h[i];
 //determinam inaltimea maxima si inaltimea minima
 max=min=h[0];
 for (i=1; i<n; i++)
     if (max<h[i]) max=h[i];
        else
        if (min>h[i]) min=h[i];
 //determinam numarul de perechi
 nr=0;
 for (i=1; i<n; i++)
     if (h[i]==max && h[i-1]==min ||
         h[i]==min && h[i-1]==max)
         nr++;
 cout<<nr<<endl;
 return 0; }
```

Exerciţiu

Modificaţi programul precedent astfel încât să realinieze elevii pentru a obţine un număr maxim de perechi de elevi alăturaţi cu proprietatea din enunţ.

Perechi

Construiţi un algoritm eficient care să determine toate perechile de numere naturale a, b, cu a≤b, având proprietatea că nu au nici o cifră comună şi suma lor este egală cu S. Valoarea S este un număr natural citit de la tastatură (S<100000000). Fiecare pereche se va scrie pe un rând al ecranului, cu un spaţiu între elementele ce compun perechea.

De exemplu, pentru S=14, se vor afişa (nu neapărat în această ordine) perechile:

```
3 11
6 8
4 10
5 9
0 14
```

(variantă Bacalaureat, august 2001)

Soluţie

O primă idee este de a genera toate perechile de numere mai mici sau egale cu S şi pentru fiecare pereche să verificăm proprietatea cerută.

Pentru a testa dacă numerele a şi b au cifre distincte, vom determina mulţimea cifrelor numărului a, apoi vom determina mulţimea cifrelor numărului b, apoi vom verifica dacă cele două mulţimi au elemente comune.

Pentru a determina mulţimea cifrelor numărului a vom considera un vector cifrea cu 10 componente (câte una pentru fiecare cifră din baza 10), având semnificaţia cifrea[i]=1, dacă cifra i apare în numărul a, şi 0, în caz contrar. Iniţial, toate componentele vectorului au valoarea 0. Vom extrage succesiv cifrele numărului a, reţinând pentru fiecare cifră extrasă valoarea 1 pe poziţia corespunzătoare cifrei din vectorul cifrea. În mod analog procedăm şi pentru numărul b, reţinând mulţimea cifrelor în vectorul cifreb. După ce am construit cele două mulţimi de cifre, le vom parcurge simultan verificând că nu există nici o cifră care să apară atât în a cât şi în b (adică să nu existe nici o poziţie k astfel încât cifrea[k] şi cifreb[k] să fie simultan 1).

```
#include <iostream.h>
int main()
{long int s, a, b, aux;
 int k, cifrea[10], cifreb[10];
 cout<<"s= "; cin>>s;
 for (a=0; a<s; a++)
      for (b=a+1; b<=s; b++)
          if (a+b==s)
```

```
            {//determinam multimea cifrelor lui a
            for (k=0; k<10; k++) cifrea[k]=0;
            aux=a;
            do {cifrea[aux%10]=1; aux/=10;} while (aux);
             //determinam multimea cifrelor lui b
            for (k=0; k<10; k++) cifreb[k]=0;
            aux=b;
            do {cifreb[aux%10]=1; aux/=10;} while (aux);
             //verificam daca cifrele sunt distincte
            for (k=0;k<10 && !(cifrea[k]&&cifreb[k]);k++);
            if (k==10) cout<<a<<' '<<b<<endl;
            }
return 0;}
```

Algoritmul de mai sus execută `s*(s-1)/2` operații, operațiile fiind relativ complexe (determinare de mulțimi de cifre și verificare). Valoarea maximă a variabilei s fiind foarte mare, timpul de execuție a acestui program va fi extrem de mare.

Algoritmul precedent poate fi transformat cu ușurință într-un algoritm liniar (care execută `s/2` operații), dacă observăm că valoarea lui `b` este unic determinată de valoarea lui `a` (`b=s-a`).

```
#include <iostream.h>
int main()
{long int s, a, b, aux;
 int k, cifrea[10], cifreb[10];
 cout<<"s= "; cin>>s;
 for (a=0; a<s/2; a++)
     {b=s-a;
      //determinam multimea cifrelor lui a
      for (k=0; k<10; k++) cifrea[k]=0;
      aux=a;
      do {cifrea[aux%10]=1; aux/=10;} while (aux);
      //determinam multimea cifrelor lui b
      for (k=0; k<10; k++) cifreb[k]=0;
      aux=b;
      do {cifreb[aux%10]=1; aux/=10;} while (aux);
      //verificam daca cifrele sunt distincte
      for (k=0; k<10 && !(cifrea[k] && cifreb[k]); k++);
      if (k==10) cout<<a<<' '<<b<<endl;
      }
return 0;}
```

Observați că `a` variază numai până la `s/2` pentru a nu genera de două ori aceeași pereche (de exemplu, și perechea `0 14`, și perechea `14 0`).

Repetiție

Se citește de la tastatură o succesiune de litere mici terminată cu caracterul punct. Să se afișeze pe ecran literele care se repetă.

Soluție

Vom citi succesiv caractere de la tastatură până la întâlnirea caracterului punct. Caracterele citite vor fi memorate în vectorul `s`, iar în variabila `n` vom reține numărul lor.

O primă idee ar fi să parcurgem vectorul `s` și pentru fiecare caracter din vector să verificăm dacă mai apare sau nu în vector pe o altă poziție și, dacă da, să îl afișăm.

```
#include <iostream.h>
#define NMax 100
int main()
{char s[NMax], c;
 int n, i, j, serepeta=1;
 n=0; //numarul de litere din secventa
 do
  {cin>>c;
   if (c!='.') s[n++]=c;   }
 while (c!='.');
 for (i=0; i<n; i++)
     {serepeta=0;
      for (j=0; j<i && !serepeta; j++)
          if (s[i]==s[j]) {cout<<s[i]; serepeta=1;}
      }
 cout<<endl; return 0; }
```

Acest program este *greșit* deoarece, în cazul în care există litere care se repetă de mai mult de două ori, ele vor fi afișate pe ecran de mai multe ori. Deci este obligatoriu să știm dacă o literă a mai fost sau nu afișată.

Ar trebui să utilizăm un vector `uz` cu 26 de componente (câte una pentru fiecare literă mică din alfabet), cu semnificația `uz[i]=1`, dacă a `i`-a literă din alfabet a mai fost afișată, și `0`, în caz contrar.

Ideea vectorului `uz` ar putea fi exploatată mai eficient. Am putea chiar de la citirea caracterelor să numărăm în `uz` de câte ori apare fiecare literă din alfabet în șir. Apoi să parcurgem vectorul `uz` și să afișăm numai literele pentru care numărul de apariții `>1`. Cu această idee am transformat un program greșit și ineficient (fiindcă executa aproximativ `n*(n-1)` operații) într-un program corect și eficient (care execută `n+26` operații).

```
#include <iostream.h>
int uz[26];
int main()
{char c;
 int i;
 do
  {cin>>c;
   if (c!='.') //numar o aparitie a literei c
      uz[c-'a']++;   }
 while (c!='.');
```

```
for (i=0; i<26; i++)
     if (uz[i]>1) cout<<(char)(i+'a');
cout<<endl; return 0; }
```

Eliminare

Se citeşte de la tastatură o secvenţă formată din maxim 100 de litere mici, până la întâlnirea caracterului punct. Dacă există în secvenţa citită litere alăturate egale, aceste litere vor fi eliminate. Dacă în urma eliminării se obţin din nou litere alăturate egale, se elimină şi acestea, eliminările efectuându-se până când în secvenţă nu există litere alăturate egale.

De exemplu, să presupunem că am citit secvenţa aacbooobcaxa. La o primă parcurgere a secvenţei, identificăm subsecvenţa aa şi subsecvenţa ooo formată din litere consecutive egale. După eliminarea acestora, secvenţa devine cbbcaxa. S-a format o nouă secvenţă de litere consecutive egale, bb. După eliminarea acestei secvenţe, obţinem ccaxa. S-a format o nouă secvenţă de litere consecutive egale, cc, pe care o eliminăm şi obţinem axa. În acest moment procesul de eliminare se opreşte, deoarece nu mai există litere consecutive egale.

Soluţie

Vom citi succesiv caractere de la tastatură până la întâlnirea caracterului punct. Caracterele citite vor fi memorate în vectorul s, iar în variabila n vom reţine numărul lor. Vom parcurge vectorul s pentru a identifica o pereche de caractere egale situate pe poziţii consecutive. Dacă o astfel de pereche a fost identificată, reţinem poziţia de început a unei secvenţe ce urmează a fi eliminată şi determinăm sfârşitul secvenţei. Eliminăm apoi secvenţa identificată, prin deplasarea la stânga a tuturor elementelor din vectorul s situate după poziţia de sfârşit a secvenţei care se elimină, urmată de micşorarea numărului de elemente din vector cu lungimea secvenţei eliminate. Procedeul de eliminare se repetă cât timp în vectorul s se produc schimbări.

```
#include <iostream.h>
#define NMax 100
int main()
{char s[NMax], c;
 int n, i, j, k, schimb=1;
 n=0; //numarul de litere din secventa
 do
  {cin>>c;
   if (c!='.') s[n++]=c;        }
 while (c!='.');
 while (schimb)
      {for (schimb=i=0; i<n-1; )
          if (s[i]==s[i+1])
             //incepe o secventa de litere egale
```

```
            {//determin sfarsitul acestei secvente
             for (j=i+1;j<n && s[j]==s[i]; j++);
             //o elimin
             for (k=j; k<n; k++) s[i+k-j]=s[k];
             //actualizez numarul de elemente din secventa
             n-=j-i;
             schimb=1;          }
           else i++; //trec mai departe
      }
for (i=0; i<n; i++) cout<<s[i];
cout<<endl; return 0; }
```

Permutare ciclică

Fie a un vector cu n componente întregi şi p un număr natural. Numim permutare ciclică dreapta cu p paşi operaţia prin care toate elementele vectorului sunt deplasate cu p poziţii spre dreapta, ultimele p elemente din vector fiind „împinse" circular pe primele p poziţii.

De exemplu, să considerăm că vectorul a conţine numerele naturale de la 0 la 7. După o deplasare ciclică dreapta cu 3 poziţii, elementelor vectorului a vor fi 5, 6, 7, 0, 1, 2, 3, 4.

Scrieţi un program care să execute o permutare ciclică dreapta cu p paşi a vectorului a.

Soluţie

O primă idee ar fi să executăm p permutări ciclice cu o poziţie spre dreapta.

```
#include <iostream.h>
int main()
{int a[100], n, p, i, aux, j;
 cout<<"n, p= "; cin>>n>>p;
 cout<<"elementele vectorului: ";
 for (i=0; i<n; i++) cin>>a[i];
 for (i=1; i<=p; i++)
     {aux=a[n-1];
      for (j=n-1; j>0; j--) a[j]=a[j-1];
      a[0]=aux;      }
 for (i=0; i<n; i++) cout<<a[i]<<' '; cout<<endl;
 return 0; }
```

Acest algoritm efectuează p parcurgeri ale vectorului, deci p*n operaţii. O metodă rapidă de permutare ciclică dreapta cu p paşi este:

- inversăm elementele vectorul din segmentul 0..n-p-1;
- inversăm elementele vectorul din segmentul n-p..n-1;
- inversăm întregul vector.

```
#include <iostream.h>
int main()
{int a[100], n, p, i, aux, j;
 cout<<"n, p= "; cin>>n>>p;
 cout<<"elementele vectorului: ";
 for (i=0; i<n; i++) cin>>a[i];
 //inversam elementele vectorului de la 0 la n-p-1
 for (i=0, j=n-p-1; i<j; i++,j--)
     { aux=a[i]; a[i]=a[j]; a[j]=aux;}
 //inversam elementele vectorului de la n-p la n-1
 for (i=n-p, j=n-1; i<j; i++,j--)
     { aux=a[i];a[i]=a[j]; a[j]=aux;}
 //inversam intreg vectorul
 for (i=0, j=n-1; i<j; i++,j--)
     { aux=a[i]; a[i]=a[j]; a[j]=aux;}
 for (i=0; i<n; i++) cout<<a[i]<<' '; cout<<endl;
 return 0; }
```

Exercițiu

Modificați programul precedent astfel încât să execute o permutare ciclică stânga cu p pași. Numim permutare ciclică stânga cu p pași operația prin care toate elementele vectorului sunt deplasate cu p poziții spre stânga, primele p elemente din vector fiind „împinse" circular pe ultimele p poziții.

Generarea mulțimii

Să considerăm mulțimea A constituită numai din elemente obținute după următoarele reguli:

1. 1∈A.
2. Dacă x∈A, atunci 2*x+1∈A.
3. Dacă x∈A, atunci 3*x+1∈A.

Scrieți un program care să genereze cele mai mici n elemente ale mulțimii A.

Soluție

Vom genera în ordine crescătoare elementele mulțimii într-un vector A. Vom inițializa primul element al vectorului A cu 1. Plecând de la 1 putem genera elementele 2*1+1=3 și 3*1+1=4.

Deci, după primul pas, elementele vectorului A sunt {1, 3, 4}. Am marcat îngroșat ultimele elemente generate.

Aplicând regulile 2 și 3 pentru ultimele elemente generate, obținem elementele 2*3+1, 3*3+1, 2*4+1 și 3*4+1. Prin urmare, componența vectorului A la pasul 2 devine {1, 3, 4, **7, 10, 9, 13**}.

La pasul următor aplicăm regulile 2 şi 3 ultimelor elemente generate (cele marcate îngroşat) şi obţinem elementele `2*7+1, 3*7+1, 2*10+1, 3*10+1, 2*9+1, 3*9+1, 2*13+1, 3*13+1`. Vectorul `A` conţine {1, 3, 4, 7, 10, 9, 13, **15, 22, 21, 31, 19, 28, 27, 40**}.

La pasul `x` aplicăm regulile 2 şi 3 ultimelor elemente generate la pasul `x-1`. Prin urmare, la pasul `x` se obţin 2^{x-1} elemente noi. Procedeul se repetă până când în vector obţinem `n` elemente. Problema este că, generând elementele în acest mod, este posibil să obţinem elemente care se repetă, prin urmare, ar trebui ca la fiecare element nou generat să verificăm dacă nu cumva există deja în vector. În al doilea rând, elementele generate nu sunt în ordine crescătoare. Deci la sfârşit trebuie să sortăm vectorul şi apoi să îl afişăm.

Algoritmul prezentat mai sus este relativ dificil de implementat şi, în plus, este şi neeficient (execută aproximativ n^2 operaţii).

O idee mai bună ar fi să generăm de la început elementele vectorului în ordine crescătoare. Pentru aceasta, la pasul `x` nu vom mai plasa în vector 2^{x-1} elemente, ci vom plasa un singur element (cel mai mic dintre cele care urmează să fie generate). Pentru a determina cel mai mic element dintre cele care urmează a fi generate nu este necesar să le determinăm pe toate, apoi să calculăm minimul. Este suficient să reţinem permanent două elemente: următorul element generat pe baza regulii 2 şi următorul element generat pe baza regulii 3. Minimul va fi selectat dintre aceste două elemente. Pentru aceasta vor fi necesari doi indici (`poz2` şi `poz3`) cu semnificaţia `poz2` = poziţia următorului element pentru care se aplică regula 2, iar `poz3` = poziţia următorului element pentru care se aplică regula 3.

```
#include <iostream.h>
int main()
{int n, poz2, poz3, A[1000], i, min;
 cout<<"n= "; cin>>n;
 A[0]=1; poz2=poz3=0;
 for (i=1; i<n; i++)
     {min=2*A[poz2]+1;
      if (min>3*A[poz3]+1) min=3*A[poz3]+1;
      A[i]=min;
      if (min==2*A[poz2]+1) poz2++;
      if (min==3*A[poz3]+1) poz3++;      }
for (i=0; i<n; i++) cout<<A[i]<<' '; cout<<endl;
return 0; }
```

Să urmărim execuţia acestui algoritm pentru `n=7`:

Pas	poz2	poz3	Vectorul A	Explicaţie
0	0	0	{1}	Iniţializăm vectorul `A` cu 1.
1	1	0	{1, 3}	Determinăm minimul dintre `2*A[poz2]+1=3` şi `3*A[poz3]+1=4`. Minimul este 3, îl plasăm în vectorul `A` şi incrementăm `poz2`.

2	1	1	{1, 3, 4}	Determinăm minimul dintre `2*A[poz2]+1=7` și `3*A[poz3]+1=4`. Minimul este 4, îl plasăm în vectorul `A` și incrementăm `poz3`.
3	2	1	{1, 3, 4, 7}	Determinăm minimul dintre `2*A[poz2]+1=7` și `3*A[poz3]+1=9`. Minimul este 7, îl plasăm în vectorul `A` și incrementăm `poz2`.
4	3	1	{1, 3, 4, 7, 9}	Determinăm minimul dintre `2*A[poz2]+1=9` și `3*A[poz3]+1=10`. Minimul este 9, îl plasăm în vectorul `A` și incrementăm `poz2`.
5	3	2	{1, 3, 4, 7, 9, 10}	Determinăm minimul dintre `2*A[poz2]+1=13` și `3*A[poz3]+1=10`. Minimul este 10, îl plasăm în vectorul `A` și incrementăm `poz3`.
6	4	2	{1, 3, 4, 7, 9, 10, 13}	Determinăm minimul dintre `2*A[poz2]+1=13` și `3*A[poz3]+1=25`. Minimul este 13, îl plasăm în vectorul `A` și incrementăm `poz2`.

Pentru `n=7`, nu am detectat cazul în care `2*A[poz2]+1=3*A[poz3]+1`. În acest caz, elementul se plasează în vector o singură dată, dar se incrementează atât `poz2`, cât și `poz3`.

Schema lui Horner

Fie $P(X)=p_0 \cdot X^n+p_1 \cdot X^{n-1}+...+p_{n-1} \cdot X+p_n$ un polinom de gradul `n` ($n \leq 100$) cu coeficienți reali. Scrieți un program care să calculeze câtul și restul împărțirii polinomului `P(X)` la `(X-a)`, unde `a` este un număr real dat.

Soluție

Vom reține coeficienții polinomului într-un vector, în ordinea descrescătoare a gradelor. Vom obține coeficienții câtului și restul în același vector, parcurgând[26] vectorul începând cu poziția `1` (primul coeficient rămâne neschimbat) și adăugând la fiecare coeficient precedentul său înmulțit cu `a`.

De exemplu, că considerăm polinomul de gradul 3 $P(X)=X^3-3X^2+1$ și `a=2`. Să urmărim pas cu pas execuția acestui algoritm:

Pas	Vectorul `P`	Explicație
0	1, -3, 0, 1	Inițializăm vectorul cu coeficienții polinomului în ordinea descrescătoare a gradelor.
1	1, -1, 0, 1	`P[1]=P[0]*a+P[1]=1*2-3=-1`
2	1, -1, -2, 1	`P[2]=P[1]*a+P[2]=-1*2+0=-2`
3	1, -1, -2, -3	`P[3]=P[2]*a+P[3]=-2*2+1=-3`

26. Metoda este cunoscută de la matematică.

Observăm că P[3] conține restul împărțirii polinomului la polinomul (X-2), iar (1, -1, -2) sunt coeficienții câtului.

```
#include <iostream.h>
int main ()
{ float P[100], a;
  int n, i;
  cout <<"n="; cin >> n;                          //citire
  for (i=0; i<=n; i++) {cout <<"P"<<i<<"="; cin>>P[i];}
  cout <<"a="; cin >> a;
  for (i=1; i<=n; i++)          //aplic schema lui Horner
      P[i]=P[i-1]*a+P[i];
  cout <<"Coeficientii catului sunt ";          //afisare
  for (i=0; i<n; i++)  cout <<P[i]<<" ";
  cout <<"\nRestul este "<<P[n]<<endl;
  return 0;}
```

Subsecvență de sumă maximă

Fie $x=(x_0, x_1, ..., x_{n-1})$ o secvență de numere întregi, dintre care cel puțin un element este pozitiv. Să se determine o subsecvență de elemente consecutive $x_i, x_{i+1}, ..., x_j$, astfel încât suma elementelor din subsecvență să fie maximă. De exemplu, pentru x=(1, 10, 3, -5, 4, 2, -100, 30, 15, 25, -10, 40, 1000), subsecvența de sumă maximă începe la poziția 7, are lungimea 6, iar suma elementelor este 1100.

Soluția 1

Considerăm toate subsecvențele de elemente consecutive, calculăm suma elementelor din subsecvență și reținem poziția de început și lungimea subsecvenței de sumă maximă.

```
#include <iostream.h>
int main ()
{int x[50], n, i, st, dr, poz, Lg, Sum, SMax;
 cout <<"n="; cin >> n;
 for (i=0; i<n; i++) {cout <<"x["<<i<<"]="; cin>>x[i];}
 for (SMax=x[0], st=0; st<n; st++)
     for (dr=st; dr<n; dr++)
         {for (Sum=0,i=st; i<=dr; i++) Sum += x[i];
              if (SMax<Sum) SMax=Sum, Lg=dr-st+1, poz=st; }
cout <<"Poz= "<<poz<<" ;Lg= "<<Lg<< " ; Smax= "<<SMax;
return 0;}
```

Datorită celor trei cicluri for imbricate, algoritmul realizează un număr de operații comparabil (ca ordin de mărime) cu n^3.

Soluția 2

Pentru a calcula suma elementelor din subsecvența de la `st` la `dr`, vom folosi suma deja calculată a elementelor de la `st` la `dr-1`. Secvența de instrucțiuni care rezolvă cerințele problemei devine:

```
for (SMax=x[0], st=0; st<n; st++)
     for (Sum=0,dr=st; dr<n; dr++)
          {Sum += x[dr];
           if (SMax < Sum) SMax=Sum, poz=st, Lg=dr-st+1;}
```

În acest caz, numărul de operații efectuate de algoritm este comparabil (ca ordin de mărime) cu n^2. Spunem că algoritmul este *pătratic*.

Soluția 3

O dată cu parcurgerea vectorului `x` de la stânga la dreapta, se va memora suma maximă calculată până la momentul curent în variabila `SMax`. În variabila `Sum` este reținută suma elementelor subsecvenței curente. Elementul `x[i]` va fi înglobat în subsecvența curentă dacă `Sum+x[i]`≥ 0. Dacă `Sum+x[i]`< 0, înglobarea lui `x[i]` în orice subsecvență ar conduce la o subsecvență cu suma elementelor mai mică decât cea maximă până la momentul curent, prin urmare, `x[i]` trebuie ignorat, iar noua subsecvență curentă va începe la poziția următoare. Mai exact, la pasul `i`: `Sum=max{Sum, 0}+x[i]; SMax=max{SMax, Sum}`. Secvența de instrucțiuni care rezolvă cerințele problemei devine:

```
for (SMax=Sum=x[0],st=poz=0, Lg=i=1; i<n; i++)
     if (Sum < 0)     //noua subsecventa incepe pe pozitia i
         Sum = x[i], st=i;
         else         //inglobez x[i] in subsecventa curenta
         {Sum += x[i];
          if (SMax<Sum) //subsecventa curenta este maximala
             SMax=Sum, poz=st, Lg=i-st+1;}
```

În acest caz, numărul de operații efectuate de algoritm este comparabil (ca ordin de mărime) cu `n`. Spunem că algoritmul a devenit *liniar*.

Cărți

Vasile joacă un joc foarte interesant. El are un pachet de `N` ($1 \leq N \leq 20000$) cărți de joc (numerotate distinct de la `1` la `N`). Cărțile din pachet sunt amestecate.

Vasile se uită la fiecare carte din pachet începând cu prima, până ajunge la cartea cu numărul `1`, pe care o scoate din pachet. Apoi caută cartea cu numărul `2`, cartea cu numărul `3` ș.a.m.d. De fiecare dată începe căutarea de unde a rămas (de la cartea care urmează după ultima carte scoasă din pachet). De fiecare dată când ajunge la sfârșitul pachetului, Vasile bate din palme și reia căutarea de la începutul pachetului. Când ultima carte din pachet este eliminată, jocul se termină.

Scrieți un program care să determine de câte ori bate din palme Vasile în timpul unui joc.

De exemplu, pentru N=3 şi ordinea cărţilor 2, 1, 3 programul va afişa 1. Pentru N=7 şi ordinea cărţilor 3, 6, 7, 1, 5, 4, 2 programul va afişa 3. Programul va citi datele din fişierul carti.in şi va afişa rezultatul în fişierul carti.out.

Soluţie

O primă idee ar fi să reţinem într-un vector cărţile din pachet în ordinea în care acestea apar în pachet şi să simulăm operaţiile pe care le face Vasile, numărând de câte ori luăm de la capăt parcurgerea pachetului de cărţi.

Această idee este mult prea costisitoare ca timp (în cazul cel mai defavorabil, adică atunci când cărţile sunt sortate descrescător, numărul de operaţii este comparabil ca ordin de mărime cu n^2).

O altă idee ar fi ca pentru fiecare carte x (de la 1 la N) să reţinem în vectorul poz poziţia sa în pachetul de cărţi. Numărul de bătăi din palme este egal cu numărul de inversiuni din vectorul poz (numărul de elemente pentru care poz[i]>poz[i+1]). În acest mod, soluţia a devenit liniară (numărul de operaţii este comparabil ca ordin de mărime cu n), esenţial pentru valori mari ale lui n.

```
#include <fstream.h>
#define MAX_N 20000
int main()
{ int i, n, x, poz[MAX_N], rez=0;
  ifstream fin("carti.in");
  fin>>n;
  for (i = 0; i<n; i++)
      {fin>>x; poz[x-1]=i;  }
  fin.close();
  for (i = 0; i<n-1; i++)
      if (poz[i] > poz[i+1]) rez++;
  ofstream fout("carti.out");
  fout<<rez<<endl;
  fout.close();
  return 0; }
```

Marcare

Numerele de la 1 la N sunt aşezate în ordine crescătoare pe circumferinţa unui cerc, astfel încât N ajunge lângă 1. Începând cu numărul S (1≤S≤N) se marchează numerele din K în K, în ordinea lor crescătoare, până când un număr este marcat de două ori (1≤N, K≤1000). Scrieţi un program care să determine numerele marcate în ordinea marcării lor, precum şi câte numere au rămas nemarcate.

De exemplu, pentru N=8, S=2 şi K=5, programul va afişa:

```
Numerele marcate in ordinea marcarii lor: 2 7 4 1 6 3 8 5
Numarul de valori nemarcate: 0
```

Soluție

Problema se rezolvă matematic: la pasul `i` se marchează numărul `x=(s+i*k)%n+n*((s+i*k)%n==0)`.

Vom memora toate valorile marcate într-o mulțime. Pentru a testa eficient dacă un număr aparține mulțimii de numere marcate, vom implementa mulțimea valorilor marcate prin vector caracteristic.

Fie `M` o mulțime de numere naturale ≤`N`. Vectorul caracteristic corespunzător mulțimii `M` este un vector cu `N+1` elemente având componentele `0` sau `1`: `M[i]=1`, dacă numărul `i` aparține mulțimii `M`, și `0`, în caz contrar.

```
#include <iostream.h>
#define NMax 1001
int M[NMax];
int main()
{int n, s, k, i;
 cout<<"n, s, k= "; cin>>n>>s>>k;
 cout<<"Numerele marcate in ordinea marcarii lor: ";
 for (i=0; M[(s+i*k)%n + n*((s+i*k)%n==0)]==0; i++)
      {//scriu elementul care urmeaza sa fie marcat
       cout<<(s+i*k)%n + n*((s+i*k)%n==0)<<' ';
       //il retin in multimea de elemente marcate
       M[(s+i*k)%n + n*((s+i*k)%n==0)]=1;}
 cout<<endl<<"Numarul de valori nemarcate: "<<n-i<<endl;
 return 0; }
```

Observație

Am declarat vectorul `M` global, pentru a fi automat inițializat cu `0` (ceea ce corespunde mulțimii vide).

Depozit

Considerăm un depozit cu `n` (`n≤1000`) camere, care conțin respectiv cantitățile de marfă C_1, C_2, ..., C_n (numere naturale). Scrieți un program care să determine un grup de camere cu proprietatea că suma cantităților de marfă pe care le conțin se poate împărți exact la cele `n` camioane care o transportă.

(Olimpiada Națională de Informatică, Iași, 1993)

Soluție

Problema este particulară: solicită determinarea unei submulțimi a unei mulțimi cu `n` elemente, divizibilă tot cu `n`. Vom construi sumele `S1`, `S2`, ..., `Sn` și resturile pe care acestea le dau prin împărțire la `n` astfel:

```
S1=C1 ⇒ R1=S1 % n
S2=C1+C2     ⇒ R2=S2 % n
...
Si=C1+C2+...+Ci    ⇒ Ri=Si % n
...
```

Cazul I: Există un rest `Ri=0`. În acest caz, suma `Si` este divizibilă cu `n`, prin urmare camerele solicitate sunt `1, 2, ..., i`.

Cazul II: Toate resturile sunt diferite de `0`. Prin urmare `R1, R2, ..., Rn` sunt `n` resturi care iau valori în mulțimea `{1, 2, ..., n-1}`. În mod obligatoriu există cel puțin două resturi egale: `Ri=Rj (i<j)`, adică `Si` și `Sj` produc același rest la împărțirea cu `n` ⇒ suma `Sj-Si` este divizibilă cu `n`, deci camerele solicitate sunt `i+1, i+2, ..., j`.

Observații

1. Soluția nu este unică. Procedeul prezentat produce o soluție posibilă.
2. Rezolvarea se bazează pe *Principiul cutiei al lui Dirichlet*[27].

```
#include <iostream.h>
int main ()
{ int SR[100], n, i, j, k, c;
  cout <<"n="; cin >> n;
  SR[0]=0;                    //calculez sumele de la citire
  for (i=1; i<=n; i++)
      {cout<<"C"<<i<<"="; cin>>c;
       SR[i]=SR[i-1]+c;}
  for (i=1; i<=n; i++) SR[i]%=n;        //calculez resturile
  cout << "Solutia este " << endl;
  for (i=1; i<=n; i++) //caut un rest = 0
      if (!SR[i])
         {for (k=1;k<=i; k++) cout <<k<<' '; return 0;}
  //cazul II, caut doua resturi egale
  for (i=1; i<n; i++)
      for (j=i+1; j<=n; j++)
          if (SR[i]==SR[j])
             {for (k=i+1;k<=j;k++) cout<<k<<' ';return 0;}
return 0;}
```

Problema celebrității

Numim celebritate o persoană care este cunoscută de toată lumea, dar nu cunoaște pe nimeni. Cunoscând relațiile dintr-o comunitate de `n` persoane (`n≤50`), se pune problema de a identifica în comunitate o celebritate, dacă aceasta există.

Soluție 1

Reprezentăm relațiile din cadrul comunității ca o matrice pătratică cu `n` linii:

`a[i][j]=1`, dacă persoana `i` cunoaște pe persoana `j`, și `0`, în caz contrar.

27. Matematicianul Peter Gustav Lejeune Dirichlet a enunțat următorul principiu: Se consideră n obiecte care trebuie plasate în p cutii, unde n>p*k, k∈N*. Oricum am plasa obiectele, există o cutie care va conține k+1 obiecte.

O primă soluție ar fi să calculăm pentru fiecare persoană p din grup numărul de persoane pe care p le cunoaşte (out) şi numărul de persoane care cunosc persoana p (in). Dacă găsim o persoană pentru care out=1 şi in=n, aceasta va fi celebritatea căutată (am presupus că orice persoană se cunoaşte pe sine însăşi).

```
#include <iostream.h>
int main()
{int a[100][100], n, i, j, in, out, p, celebritate=-1;
 cout <<"n="; cin >> n;
 cout << "Introduceti matricea relatiilor!\n";
 for (i=0; i<n; i++)
     for (j=0; j<n; j++) cin>>a[i][j];
 for (p=0; p<n; p++)
     {for(in=out=j=0; j<n; j++)
          in+=a[j][p], out+=a[p][j];
     if (in==n && out==1) celebritate=p;}
if (celebritate<0) cout<<"In grup nu exista celebritati !";
    else cout<<celebritate+1<<" este o celebritate. ";
return 0;}
```

Se poate observa cu uşurinţă că algoritmul este *pătratic* (numărul de operaţii efectuate de algoritm este comparabil ca ordin de mărime cu n^2).

Soluţie 2

Putem îmbunătăţi algoritmul făcând observaţia că, atunci când testăm relaţiile dintre două persoane x şi y, apar următoarele posibilităţi :

a[x,y]=0, şi în acest caz y nu are nici o şansă să fie celebritate, sau

a[x,y]=1, şi în acest caz x nu poate fi celebritate.

Deci la un test eliminăm o persoană care nu are şanse să fie celebritate. Efectuând succesiv n-1 teste, în final vom avea o singură persoană candidat la celebritate. Rămâne să calculăm numărul de persoane cunoscute şi numărul de persoane care îl cunosc pe acest candidat, singura celebritate posibilă. Secvenţa de determinare a celebrităţii devine:

```
for (celebritate=0, i=1; i<n; i++)
    if (a[celebritate][i]) celebritate=i;
for (out=in=i=0; i<n; i++)
    in+=a[i][celebritate], out+=a[celebritate][i];
```

În acest caz, algoritmul a devenit *liniar* (numărul de operaţii efectuate de algoritm este comparabil ca ordin de mărime cu n).

Cadrane

Fie n un număr natural impar (n≤100). Să se construiască o matrice pătratică având n linii şi n coloane după modelul din exemplul următor:

2	2	0	1	1
2	2	0	1	1
0	0	0	0	0
3	3	0	4	4
3	3	0	4	4

Soluție

Numărul n fiind impar, există o linie și o coloană „de mijloc". Dacă vom numerota liniile și coloanele de la 0 la n-1, linia, respectiv coloana din mijloc, are numărul n/2. Observăm din exemplu că toate elementele de pe linia, respectiv coloana „de mijloc", au valoarea 0.

Putem să ne imaginăm că elementul din mijlocul matricei (cel de coordonate n/2, n/2) ar fi centrul unui sistem de coordonate, linia n/2 fiind axa Ox, iar coloana n/2 fiind axa Oy. Observăm din exemplu că celelalte elemente ale matricei au valori egale cu numărul cadranului în care se află.

```
#include <iostream.h>
#define NMax 100
int main()
{int a[NMax][NMax], n, i, j;
 cout<<"n= "; cin>>n;
 //completam linia n/2 cu 0
 for (i=0; i<n; i++) a[n/2][i]=0;
 //completam coloana n/2 cu 0
 for (i=0; i<n; i++) a[i][n/2]=0;
 //completam cadranul 1
 for (i=0; i<n/2; i++)
     for (j=n/2+1; j<n; j++) a[i][j]=1;
 //completam cadranul 2
 for (i=0; i<n/2; i++)
     for (j=0; j<n/2; j++) a[i][j]=2;
 //completam cadranul 3
 for (i=n/2+1; i<n; i++)
     for (j=0; j<n/2; j++) a[i][j]=3;
 //completam cadranul 4
 for (i=n/2+1; i<n; i++)
     for (j=n/2+1; j<n; j++) a[i][j]=4;
 //afisam matricea
 for (i=0; i<n; i++)
     {for (j=0; j<n; j++) cout<<a[i][j]<<' ';
      cout<<endl;     }
 return 0; }
```

Matrice

Fie n un număr natural impar (n≤100). Să se construiască o matrice pătratică având n linii și n coloane după modelul din exemplul următor:

0	1	1	1	0
2	0	1	0	4
2	2	0	4	4
2	0	3	0	4
0	3	3	3	0

Soluție

Din exemplu, observăm că toate elementele situate pe diagonala principală sau pe diagonala secundară sunt 0. Cele două diagonale împart matricea în patru zone. Zona 1 este constituită din elemente situate deasupra diagonalei principale și deasupra diagonalei secundare. Toate elementele din zona 1 au valoarea 1. Zona 2 este constituită din elemente situate sub diagonala principală și deasupra diagonalei secundare. Toate elementele din zona 2 au valoarea 2. Zona 3 este constituită din elemente situate sub diagonala principală și sub diagonala secundară. Toate elementele din zona 3 au valoarea 3. Zona 4 este constituită din elemente situate deasupra diagonalei principale și sub diagonala secundară. Toate elementele din zona 4 au valoarea 4.

```
#include <iostream.h>
#define NMax 100
int main()
{int a[NMax][NMax], n, i, j;
 cout<<"n= "; cin>>n;
 //completam diagonala principala cu 0
 for (i=0; i<n; i++) a[i][i]=0;
 //completam diagonala secundara cu 0
 for (i=0; i<n; i++) a[i][n-i-1]=0;
 //completam zona 1
 for (i=0; i<n/2; i++)
     for (j=i+1; j<n-i-1; j++) a[i][j]=1;
 //completam zona 2
 for (j=0; j<n/2; j++)
     for (i=j+1; i<n-j-1; i++) a[i][j]=2;
 //completam zona 3
 for (i=n/2+1; i<n; i++)
     for (j=n-i; j<i; j++) a[i][j]=3;
 //completam cadranul 4
 for (j=n/2+1; j<n; j++)
     for (i=n-j; i<j; i++) a[i][j]=4;
 //afisam matricea
 for (i=0; i<n; i++)
     {for (j=0; j<n; j++)  cout<<a[i][j]<<' ';
     cout<<endl;     }
 return 0; }
```

Situație școlară

Profesorul diriginte dorește ca la sfârșitul anului școlar părinții să aibă o imagine clară asupra situației școlare a copilului lor, dar și a întregii clase.

Din catalogul clasei (memorat în fișierul `catalog.in`), dirigintele preia următoarele informații: numărul de elevi, numărul de obiecte de studiu, media anuală a fiecărui elev pentru fiecare obiect de studiu. Dacă un elev nu studiază o anumită disciplină (de exemplu, este scutit la educație fizică), media lui anuală la obiectul respectiv va fi 0 și nu va fi luată în calcul pentru determinarea mediei generale a elevului.

Profesorul diriginte trebuie să determine și să afișeze în fișierul `situatie.out` următoarele informații:
1. Media generală a fiecărui elev promovat (care nu are nici o medie mai mică decât `5`).
2. Media generală a clasei.
3. Media generală a clasei pentru fiecare obiect de studiu.
4. Elevii premianți (identificați prin numărul lor de ordine din catalog).

Regulamentul de stabilire a premiilor este următorul:
- obține premiul I elevul cu media cea mai mare din clasă, medie care nu trebuie să fie mai mică de 9.50; dacă există mai mulți elevi cu medie maximă, toți obțin premiul I.
- obține premiul al II-lea elevul (sau elevii, în caz de egalitate) cu cea mai mare medie din clasă (exceptând elevii care au obținut premiul I), medie care nu trebuie să fie mai mică decât 9.00.
- obține premiul al III-lea elevul (sau elevii, în caz de egalitate) cu cea mai mare medie din clasă (exceptând elevii care au obținut premiul I și premiul al II-lea), medie care nu trebuie să fie mai mică decât 8.50.

Soluție

Vom memora datele din catalog într-o matrice `C` cu `n` linii, numerotate de la `1` la `n` (unde `n` este numărul de elevi din clasă), și `m` coloane, numerotate de la `1` la `m` (unde `m` este numărul de obiecte de studiu). Elementul de pe linia `i` și coloana `j` din matricea `C` reprezintă media anuală pe care elevul `i` a obținut-o la obiectul `j`.

Observați că linia `0` și coloana `0` din matrice nu sunt utilizate deocamdată. Vom utiliza coloana `0` pentru a reține mediile generale ale elevilor (`C[i][0]`=media generală a elevului `i`), iar linia `0` o vom utiliza pentru a reține mediile generale pe obiecte (`C[0][j]`=media clasei la obiectul `j`). Rămâne disponibil elementul de pe linia `0` și coloana `0` (`C[0][0]`) pentru a reține media generală a clasei.

Pentru a determina media generală a fiecărui elev trebuie să parcurgem fiecare linie a matricei și să calculăm media aritmetică a elementelor diferite de `0` de pe linia

respectivă. Pe parcursul acestui calcul verificăm şi dacă elevul este promovat. Dacă vom descoperi o media mai mică decât 5, media generală a acestui elev va rămâne 0.

Pentru a determina media clasei pentru fiecare obiect trebuie să parcurgem fiecare coloană a matricei şi să calculăm media aritmetică a elementelor diferite de 0 de pe linia respectivă.

Pentru a calcula media generală a clasei este suficient să facem media mediilor generale ale elevilor promovaţi (deci media elementelor nenule de pe coloana 0).

Pentru a determina premianţii, determinăm `max1`, `max2` şi `max3` (cele mai mari trei medii generale distincte), punând condiţia suplimentară ca `max1>=9.50` (în caz contrar, nu se acordă premiul I), `max2>=9.00` (în caz contrar, nu se acordă premiul al II-lea), `max3>=8.50` (în caz contrar, nu se acordă premiul al III-lea).

Dacă se acordă premiul I, vom parcurge toţi elevii şi vom afişa pe cei care au media generală egală cu `max1`. În acelaşi mod procedăm şi cu premiile al II-lea şi al III-lea.

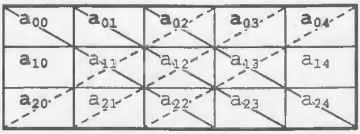
```
#include <fstream.h>
#include <iomanip.h>
int n, m;
double C[50][30];
int main()
{int i, j, nr, corigent;
 double s, max1, max2, max3;
 ifstream fin("Catalog.in");
 ofstream fout("Situatie.out");
 //citire
 fin>>n>>m;
 for (i=1; i<=n; i++)
     for (j=1; j<=m; j++) fin>>C[i][j];
 fin.close();
 //calculam mediile generale ale elevilor
 for (i=1; i<=n; i++)
     {s=0; nr=0; corigent=0;
      for (j=1; j<=m && !corigent; j++)
          if (C[i][j])
             if (C[i][j]<5) corigent =1;
                 else {s+=C[i][j]; nr++;}
      if (corigent) C[i][0]=0;
          else C[i][0]=s/nr;
     }
 //afisarile se vor face cu doua zecimale
 fout<<setprecision(2);
 //afisam  mediile generale ale elevilor
 fout<<"Mediile generale ale elevilor:\n";
 for (i=1; i<=n; i++)
     fout<<"Media elevului "<<i<<" este "<<C[i][0]<<endl;
 fout<<endl;
```

```
//calculam mediile pe obiecte
for (i=1; i<=m; i++)
    {for (s=0, nr=0, j=1; j<=n ; j++)
         if (C[j][i])
            {s+=C[j][i]; nr++;}
     C[0][i]=s/nr; . }
//afisam mediile pe obiecte
fout<<"Mediile clasei pe obiecte:\n";
for (i=1; i<=m; i++)
    fout<<"Media la obiectul "<<i<<" e "<<C[0][i]<< endl;
//calculam media generala clasei
s=0; nr=0;
for (i=1; i<=n; i++)
    if (C[i][0]){s+=C[i][0]; nr++;}
C[0][0]=s/nr;
fout<<"\nMedia generala a casei este "<<C[0][0]<<endl;
//calculam max1, max2, max3
max1=max2=max3=0;
for (i=1; i<=n; i++)
    {if (C[i][0]>max1)
        {max3=max2; max2=max1;max1=C[i][0];}
     if (C[i][0]<max1 && C[i][0]>max2)
        {max3=max2; max2=C[i][0];}
     if (C[i][0]<max2 && C[i][0]>max3) max3=C[i][0];
    }
if (max1>=9.50)
   //se acorda premiul I
   {fout<<"Se acorda Premiul I cu media "<<max1;
    fout<<" elevilor:\n";
    for (i=1; i<=n; i++)
        if (C[i][0]==max1) fout<<i<<' ';
    fout<<endl;  }
   else
   {fout<<"Nu se acorda Premiul I\n";
    max3=max2; max2=max1;}
if (max2>=9)
   //se acorda premiul al II-lea
   {fout<<"Se acorda Premiul al II-lea cu media "<<max2;
    fout<<" elevilor :\n";
    for (i=1; i<=n; i++)
        if (C[i][0]==max2) fout<<i<<' ';
    fout<<endl; }
   else
   {fout<<"Nu se acorda Premiul al II-lea\n";
    max3=max2;  }
if (max3>=8.50)
   //se acorda premiul al III-lea
   {fout<<"Se acorda Premiul al III-lea cu media "<<max3;
    fout<<" elevilor :\n";
```

```
    for (i=1; i<=n; i++)
        if (C[i][0]==max3)  fout<<i<<' ';
    fout<<endl;
   }
   else fout<<"Nu se acorda Premiul al III-lea\n";
fout<<endl<<max1<<' '<<max2<<' '<<max3<<endl;
fout.close(); return 0; }
```

Pseudodiagonale

Fie A o matrice cu n linii şi m coloane ($1 \le n < m \le 100$) cu componente întregi. Nefiind pătratică, matricea A nu are diagonale, dar are pseudodiagonale principale şi secundare, aşa cum este ilustrat în exemplul următor:



În exemplu am figurat o matrice cu 3 linii şi 5 coloane. În această matrice sunt 3 pseudodiagonale principale (trasate cu linie continuă) şi 3 pseudodiagonale secundare (trasate cu linie punctată).

Scrieţi un program care să determine suma elementelor de pe fiecare pseudodiagonală principală şi de pe fiecare pseudodiagonală secundară.

Soluţie

Pentru a rezolva această problemă sunt câteva întrebări la care trebuie să răspundem:

1. Câte pseudodiagonale există într-o matrice cu n linii şi m coloane, cu n<m?
2. Care este poziţia de început şi poziţia de sfârşit pentru fiecare pseudodiagonală?
3. Care este relaţia dintre indicele de linie şi cel de coloană pe fiecare pseudodiagonală?

Pentru a răspunde la aceste întrebări vom urmări într-un tabel variaţia poziţiei de început şi a poziţiei de sfârşit pentru pseudodiagonale:

Pseudodiagonala	Linie început	Coloana început	Linie sfârşit	Coloana sfârşit
0	0	0	n-1	n-1
1	0	1	n-1	n
2	0	2	n-1	n+1
...				...
d	0	d	n-1	n+d-1
...				
ultima	0	m-n	n-1	m-1

Observaţii

1. Fiecare pseudodiagonală începe pe linia `0` şi se termină pe linia `n-1`.
2. Pseudodiagonala `0` începe în coloana `0`, pseudodiagonala `1` începe în coloana `1` etc. Prin urmare, coloana de început a unei pseudodiagonale este egală cu numărul pseudodiagonalei.
3. Interpretând geometric, observăm că o pseudodiagonală principală formează un unghi de 45° cu orizontala, deci este ipotenuza unui triunghi dreptunghic isoscel. Pentru a calcula coloana de sfârşit pentru fiecare pseudodiagonală, punem condiţia de egalitate a catetelor.
4. Ultima pseudodiagonală se va termina în coloana `m-1`, deci (din condiţia de triunghi dreptunghic isoscel) deducem că începe în coloana `n-m`. Am determinat astfel că numărul de pseudodiagonale este `n-m+1`.

Vom parcurge succesiv cele `n-m+1` pseudodiagonale principale, calculând pentru fiecare pseudodiagonală suma elementelor.

Pentru a parcurge pseudodiagonala principală `d`, trebuie să determinăm relaţia care există între indicele de linie şi cel de coloană pe această pseudodiagonală. În acest scop, vom urmări în tabelul următor variaţia indicelui de linie în raport cu cel de coloană pe pseudodiagonala `d`:

Linia	0	1	2	...	i	...	n-1
Coloana	d	d+1	d+2		i+d		n+d-1

Observaţii

Când indicele de linie creşte cu 1, indicele de coloană creşte, de asemenea, cu 1. Pin urmare, diferenţa dintre indicele de coloană şi indicele de linie este constantă: `coloana-linie=d`. Deducem relaţia: `coloana=linie+d`.

Raţionamentul este analog pentru pseudodiagonalele secundare:

Pseudodiagonala	**Linie început**	**Coloana început**	**Linie sfârşit**	**Coloana sfârşit**
0	0	m-1	n-1	m-n
1	0	m-2	n-1	m-n-1
2	0	m-3	n-1	m-n-2
...				
d	0	m-d-1	n-1	m-n-d
...				
ultima	0	n-1	n-1	0

Observaţii

1. Fiecare pseudodiagonală secundară începe pe linia `0` şi se termină pe linia `n-1`.
2. Pseudodiagonala secundară `0` începe în coloana `m-1`, pseudodiagonala `1` începe în coloana `m-2` etc. Prin urmare, coloana de început a pseudodiagonalei secundare `d` este egală cu `m-d+1`.

3. Interpretând geometric, observăm că o pseudodiagonală secundară formează un unghi de 45° cu orizontala, deci este ipotenuza unui triunghi dreptunghic isoscel. Pentru a calcula coloana de sfârşit pentru fiecare pseudodiagonală, punem condiţia de egalitate a catetelor.
4. Ultima pseudodiagonală se va termina în coloana 0, deci (din condiţia de triunghi dreptunghic isoscel) deducem că începe în coloana n-1.

Vom parcurge succesiv cele n-m+1 pseudodiagonale secundare, calculând pentru fiecare pseudodiagonală secundară suma elementelor. Pentru a parcurge pseudodiagonala secundară d, trebuie să determinăm relaţia care există între indicele de linie şi cel de coloană pe această pseudodiagonală. În acest scop, vom urmări în tabelul următor variaţia indicelui de linie în raport cu cel de coloană pe pseudodiagonala secundară d:

Linia	0	1	2	...	i	...	n-1
Coloana	m-d-1	m-d-2	m-d-3		m-d-i-1		m-n-d

Observaţii

Când indicele de linie creşte cu 1, indicele de coloană scade cu 1. Prin urmare, suma dintre indicele de coloană şi indicele de linie este constantă: coloana+linie=m-d-1. Deducem relaţia: coloana=m-d-linie-1.

```
#include <iostream.h>
int main()
{int n, m, a[100][100], i, j, d, s;
 cout<<"n= "; cin>>n;
 cout<<"m= "; cin>>m;
 cout<<"Introduceti elementele matricei, linie cu linie\n";
 for (i=0; i<n; i++)
     for (j=0; j<m; j++) cin>>a[i][j];
 //parcurgem pseudodiagonalele principale
 for (d=0; d<m-n+1; d++)
     {s=0; //suma elementelor de pe pseudodiagonala d
      for (i=0; i<n; i++)
          s+=a[i][i+d];
      cout<<"Suma elementelor de pe pseudodiagonala
            principala "<<d<<" este "<<s<<endl;
     }
 //parcurgem pseudodiagonalele secundare
 for (d=0; d<m-n+1; d++)
     {s=0; //suma elementelor de pe pseudodiagonala d
      for (i=0; i<n; i++)
          s+=a[i][m-d-i-1];
      cout<<"Suma elementelor de pe pseudodiagonala
            secundara "<<d<<" este "<<s<<endl;
     }
 return 0;}
```

Exerciţiu

Modificaţi programul precedent astfel încât să rezolve problema pentru cazul n>m.

Submulţimi cu sume egale

Fie n un număr natural nenul, n≤50. Scrieţi un program care să determine n submulţimi disjuncte cu n elemente din mulţimea {1, 2, ..., n^2}, submulţimi pentru care suma elementelor este aceeaşi.

Soluţie

Problema pare deosebit de complexă. O primă idee, greu de implementat, este de a genera toate submulţimile de n elemente ale mulţimii {1, 2, ..., n^2}, apoi de a alege n submulţimi disjuncte cu suma elementelor aceeaşi.

Pe lângă faptul că această idee este greu de implementat, este şi inutilizabilă din punct de vedere practic, pentru că numărul de submulţimi este foarte mare.

Soluţia se bazează pe o idee constructivă.

1. Plasăm în ordine elementele de la 1 la n^2 într-o matrice pătratică cu n linii şi n coloane. De exemplu, pentru n=4 obţinem:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

2. Alegem ca primă submulţime elementele de pe diagonala principală {1, 6, 11, 16}. Suma elementelor din submulţime este S=1+6+11+16=34.

3. Pentru a construi cu uşurinţă celelalte n-1 submulţimi cu suma elementelor S, vom extinde matricea, copiind în continuare primele n-1 linii. Se obţine astfel o matrice cu 2*n-1 linii şi n coloane. De exemplu, pentru n=4 obţinem:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
1	2	3	4
5	6	7	8
9	10	11	12

Această matrice are n pseudodiagonale principale. Mulțimile elementelor de pe oricare două pseudodiagonale sunt disjuncte, iar suma elementelor de pe fiecare pseudodiagonală este aceeași. Problema se reduce astfel la o problemă asemănătoare cu precedenta.

```
#include <iostream.h>
int main()
{int n, m, a[101][51], i, j, d, nr=1;
 cout<<"n= "; cin>>n;
 m=2*n-1;
 for (i=0; i<n; i++)
     for (j=0; j<n; j++) a[i][j]=a[i+n][j]=nr++;
 //parcurgem pseudodiagonalele
 for (d=0; d<n; d++)
     {//afisez elementele de pe pseudodiagonala d
      cout<<"Submultimea "<<d+1<<": { ";
      for (j=0; j<n; j++) cout<<a[j+d][j]<<' ';
      cout<<"}\n";      }
 return 0;}
```

Dreptunghi de sumă maximă

Fie A o matrice cu n linii și m coloane cu elemente întregi (n, m≤50). Să se determine o zonă dreptunghiulară în această matrice pentru care suma elementelor din această zonă este maximă. În cazul în care există mai multe soluții, veți afișa o zonă de arie maximă.

Soluție

O zonă dreptunghiulară din matrice este formată din elemente A[i][j] cu proprietatea că x1≤i≤x2 și y1≤j≤y2. Pentru a identifica o zonă dreptunghiulară din matrice este suficient să reținem (x1, y1) (coordonatele colțului din stânga-sus al zonei) și (x2, y2) (coordonatele colțului din dreapta-jos al zonei).

O primă idee ar fi să generăm toate zonele dreptunghiulare și, pentru fiecare zonă, să calculăm suma elementelor. Obținem astfel toate sumele posibile, dar reținem numai suma maximă.

```
#include <iostream.h>
int main()
{int n, m, a[100][100], i, j, x1,y1,x2,y2;
 int smax, x1max, y1max, x2max, y2max, s;
 cout<<"n, m= "; cin>>n>>m;
 cout<<"Introduceti elementele matricei, linie cu linie\n";
 for (i=1; i<=n; i++)
    for (j=1; j<=m; j++) cin>>a[i][j];
 smax=a[0][0]; x1max=x2max=y1max=y2max=0;
 for (x1=1; x1<=n; x1++)
     for (y1=1; y1<=m; y1++)
```

```
        for (x2=x1; x2<=n; x2++)
            for (y2=y1; y2<=m; y2++)
                {
                /* calculez suma elementelor din
                dreptunghiul cu coltul stanga-sus (x1, y1)
                si coltul dreapta-jos (x2, y2) */
                for (s=0, i=x1; i<=x2; i++)
                    for (j=y1; j<=y2; j++) s+=a[i][j];
                if (s>smax)
                   {smax=s; //retin suma maxima
                    x1max=x1; y1max=y1; //si colturile
                    x2max=x2; y2max=y2;}
                  else //aceeasi suma dar arie mai mare?
                  if (s==smax && (x2-x1+1)*(y2-y1+1)>
                     (x2max-x1max+1)*(y2max-y1max+1))
                     {x1max=x1;y1max=y1; //retin colturile
                      x2max=x2; y2max=y2;}
                }
 cout<<"Suma maxima: "<<smax<<endl;
 cout<<"Coltul stanga-sus: "<<x1max<<", "<<y1max<<endl;
 cout<<"Coltul dreapta-jos: "<<x2max<<", "<<y2max<<endl;
 return 0; }
```

Acesta este cel mai simplu, dar totodată și cel mai ineficient algoritm la care ne-am fi putut gândi. Numărul de operații este aproximativ $(n*m)^3$.

O idee mai eficientă, dar care necesită spațiu de memorie suplimentar este de a calcula și memora într-o matrice auxiliară `sum` sumele din toate dreptunghiurile care au colțul din stânga-sus `(1, 1)` și colțul din dreapta-jos `(x, y)`.

Suma elementelor din orice alt dreptunghi se poate determina pe baza informațiilor din matricea `sum`. Mai exact, pentru a determina suma elementelor din dreptunghiul cu colțul stânga-sus `(x1, y1)` și cel din dreapta-jos `(x2, y2)`, formula de calcul este:

```
sum[x2][y2]-sum[x2][y1-1]-sum[x1-1][y2]+sum[x1-1][y1-1]
```

```
#include <iostream.h>
int sum[100][100];
int main()
{int n, m, a[100][100], i, j, x1,y1,x2,y2;
 int smax, x1max, y1max, x2max, y2max, s;
 cout<<"n, m= "; cin>>n>>m;
 cout<<"Introduceti elementele matricei, linie cu linie\n";
 for (i=1; i<=n; i++) for (j=1; j<=m; j++) cin>>a[i][j];
 smax=a[0][0]; x1max=x2max=y1max=y2max=0;
 for (x1=1; x1<=n; x1++)
     for (y1=1; y1<=m; y1++)
    /* calculez suma elementelor din dreptunghiul cu
    coltul stanga-sus (0,0) si coltul dreapta-jos (x1,y1)*/
          for (i=1; i<=x1; i++)
               for (j=1; j<=y1; j++) sum[x1][y1]+=a[i][j];
```

```
for (x1=1; x1<=n; x1++)
    for (y1=1; y1<=m; y1++)
        for (x2=x1; x2<=n; x2++)
            for (y2=y1; y2<=m; y2++)
                {s=sum[x2][y2]-sum[x1-1][y2]-
                  sum[x2][y1-1]+sum[x1-1][y1-1];
                if (s>smax)
                  {smax=s;
                   x1max=x1; y1max=y1;
                   x2max=x2; y2max=y2;}
                  else
                  if (s==smax &&
                     (x2-x1+1)*(y2-y1+1)>
                     (x2max-x1max+1)*(y2max-y1max+1))
                     {x1max=x1; y1max=y1;
                      x2max=x2; y2max=y2;}
                }
cout<<"Suma maxima: "<<smax<<endl;
cout<<"Coltul stanga-sus: "<<x1max<<", "<<y1max<<endl;
cout<<"Coltul dreapta-jos: "<<x2max<<", "<<y2max<<endl;
return 0; }
```

Complexitatea algoritmului în acest caz este de $(n*m)^2$. Există şi algoritmi mai performanţi de rezolvare a acestei probleme, dar îi vom studia ulterior.

Secvenţe

Fie `A` o secvenţă formată din `N` valori distincte, cuprinse între `1` şi `N`, date într-o ordine oarecare. Definim secvenţa `B` astfel: `B[k]=1` dacă şi numai dacă primele `k` valori din secvenţa `A` sunt numai numere distincte din mulţimea {`1`, `2`, ..., `k`} într-o ordine oarecare; în caz contrar, `B[k]=0`.

Dată fiind secvenţa `B` şi câteva elemente din secvenţa `A`, scrieţi un program care să reconstituie secvenţa `A`.

Date de intrare

Fişierul de intrare `secv.in` conţine:

- pe prima linie, două numere naturale `N` şi `M`;
- pe cea de-a doua linie, elementele secvenţei `B`, separate prin spaţii;
- pe fiecare dintre următoarele `M` linii, informaţii despre elementele cunoscute din secvenţa `A` sub forma a două numere naturale `X` şi `Y` cu semnificaţia „al `X`-lea element din secvenţa `A` este `Y`".

Date de ieşire

Fişierul de ieşire `secv.out` conţine o singură linie pe care sunt afişate elementele secvenţei `A` separate prin câte un spaţiu. Dacă nu există soluţie, se va afişa în fişierul de ieşire mesajul `IMPOSIBIL`.

Restricții

- $1 \le N \le 10000$
- $0 \le M \le N$
- informațiile din fișierul de intrare nu sunt contradictorii;
- datele de pe aceeași linie din fișiere sunt separate prin spații;
- soluția nu este în mod necesar unică.

Exemple

secv.in	secv.out	secv.in	secv.out
8 3 0 0 0 1 0 0 1 1 1 2 5 6 2 7	IMPOSIBIL	7 2 0 0 0 1 0 0 1 1 2 5 6	2 4 3 1 6 7 5

(Olimpiada Municipală de Informatică, clasa a IX-a, Iași, 2003)

Soluție

Vom utiliza un vector `sol` cu `N` componente în care încercăm să reconstituim secvența `A`. Chiar de la citire vom plasa în vectorul `sol` cele `M` elemente pentru care sunt cunoscute pozițiile. De asemenea, vom utiliza un vector `stiu`, cu semnificația `stiu[i]=1`, dacă elementul `i` este deja plasat în vectorul `sol`, și `0`, în caz contrar. La fiecare pas vom căuta un `1` în vectorul `B`. Segmentul din secvența `A` care ține de la precedentul `1` (sau de la începutul vectorului) până la acest `1` trebuie să conțină toate valorile întregi din intervalul dintre aceste două poziții. O parte dintre aceste valori sunt deja fixate, celelalte le plasăm în orice ordine dorim.

```
#include <fstream.h>
#define INFILE    "secv.in"
#define OUTFILE   "secv.out"
#define MAXN       10000
int b[MAXN+1], sol[MAXN+1], stiu[MAXN+1];
int n, M;
int main ( )
{ int i, temp, temp1, max;
  //citire
  ifstream fin(INFILE);
  fin>>n>>M;
  for (i=1; i<=n; i++) fin>>b[i];
  for (i=0; i<M; i++)
      {fin>>temp>>temp1;
       sol[temp]=temp1; stiu[temp1]=1;}
  fin.close();
  int curent=1, inceput, w,  j, ok=1;
  while (curent<=n && ok)
       {inceput=curent;
         while (!b[curent] && curent<=n) curent++;
```

```
        if (curent>n) ok=0;
       /*secventa de la inceput la curent este o permutare
       a valorilor de la inceput la curent*/
       for (w=inceput, j=curent; j>=inceput; j--)
          if (!stiu[j])
           {while (sol[w]) w++;
            sol[w]=j; }
     curent++;
   }
ofstream fout(OUTFILE);
if (!ok) fout<<"IMPOSIBIL\n";
   else
   {//verific
    for (max=0, i=1; i<=n && ok; i++)
        {if (sol[i]>max) max=sol[i];
         if (b[i] && max!=i) ok=0;
         if (!b[i] && max==i) ok=0;          }
    if (!ok) fout<<"IMPOSIBIL\n";
       else
       {for (i=1; i<n; i++) fout<<sol[i]<<' ';
        fout<<sol[n]<<endl;}
   }
fout.close(); return 0;}
```

Problema spectacolelor

Managerul artistic al unui festival trebuie să selecteze o mulțime cât mai amplă de spectacole ce pot fi jucate în singura sală pe care o are la dispoziție. Știind că i s-au propus $n \leq 100$ spectacole și pentru fiecare spectacol i i-a fost anunțat intervalul în care se poate desfășura $[s_i, f_i)$ (s_i reprezintă ora și minutul de început, iar f_i ora și minutul de final al spectacolului i), scrieți un program care să permită spectatorilor vizionarea unui număr cât mai mare de spectacole. De exemplu, dacă vom citi $n=5$ și următorii timpi:

```
12 30 16 30
15 0 18 0
10 0 18 30
18 0 20 45
12 15 13 0
```

Spectacolele selectate sunt: `5 2 4`.

Soluție

Ordonăm spectacolele crescător după ora de final. Selectăm inițial primul spectacol (deci cel care se termină cel mai devreme). La fiecare pas selectăm primul spectacol neselectat care nu se suprapune cu cele deja selectate (deci care începe după ce se termină ultimul spectacol selectat).

```
#include <iostream.h>
int inceput[100], sfarsit[100], nr[100];
int main()
{int n, i, h, m, schimb, ultim, aux;
 cout << "n= "; cin >> n;                                    //citire
 cout<<"Introduceti inceputul si sfarsitul spectacolelor";
 for (i=0; i<n; i++)
       {nr[i]=i+1;              //transform timpul in minute
        cin>>h>>m; inceput[i]=h*60+m;
        cin>>h>>m; sfarsit[i]=h*60+m;}
 //ordonez spectacolele crescator dupa ora de final
 do
  {schimb=0;
   for (i=0; i<n-1; i++)
        if (sfarsit[nr[i]]>sfarsit[nr[i+1]])
           {aux=nr[i];nr[i]=nr[i+1];nr[i+1]=aux; schimb=1;}
  }
 while (schimb);
 cout << "Spectacolele selectate sunt:\n"<<nr[0]<<' ';
 for (ultim=0, i=1; i<n; i++)
      if (inceput[nr[i]]>=sfarsit[nr[ultim]])
          {cout <<nr[i]<<' '; ultim=i;}
 cout<<endl;
 return 0;}
```

Pentru rezolvarea acestei probleme am utilizat o metodă de programare importantă, denumită *Greedy*. În general, metoda *Greedy* se aplică problemelor de optimizare. Specificul acestei metode constă în faptul că se construieşte soluţia optimă pas cu pas, la fiecare pas fiind selectat (sau „înghiţit") în soluţie elementul care pare „cel mai bun" la momentul respectiv, în speranţa că această alegere locală va conduce la optimul global.

Algoritmii *Greedy* sunt foarte eficienţi, dar nu conduc în mod necesar la o soluţie optimă. Şi nici nu este posibilă formularea unui criteriu general conform căruia să putem stabili exact dacă metoda *Greedy* rezolvă sau nu o anumită problemă de optimizare. Din acest motiv, orice algoritm *Greedy* trebuie însoţit de o demonstraţie a corectitudinii sale[28]. Demonstraţia faptului că o anumită problemă are proprietatea alegerii *Greedy* se face de obicei prin inducţie matematică.

Problema rucsacului

Un hoţ neprevăzător are la dispoziţie un singur rucsac cu care poate transporta o greutate maximă `Gmax`. Hoţul are de ales din $n \leq 50$ obiecte şi, evident, intenţionează să obţină un câştig maxim în urma singurului transport pe care îl poate face. Cunoscând

28. Demonstraţia depăşeşte nivelul de cunoştinţe al unui elev de clasa a IX-a.

pentru fiecare obiect i greutatea g_i şi câştigul c_i pe care hoţul l-ar obţine transportând obiectul respectiv în întregime, scrieţi un program care să determine o încărcare optimă a rucsacului, în ipoteza că hoţul poate încărca în rucsac orice parte dintr-un obiect. De exemplu, pentru n=5, GMax=100 şi câştigurile-greutăţile următoare:

(1000 120)(500 20)(400 200)(1000 100)(25 1), se va afişa pe ecran:

```
2              100.00%
4              79.00%
5              100.00%
```

Soluţie

Vom reprezenta o soluţie a problemei ca un vector x cu n componente, în care reţinem pentru fiecare obiect fracţiunea încărcată în rucsac de hoţ. Deci vectorul x trebuie să îndeplinească următoarele condiţii:

1. $x_i \in [0, 1], \forall i \in \{1, 2, ..., n\}$;
2. $g_1 \cdot x_1 + g_2 \cdot x_2 + ... + g_n \cdot x_n \leq Gmax$;
3. $f(x) = c_1 \cdot x_1 + c_2 \cdot x_2 + ... + c_n \cdot x_n$ este maximă.

Ordonăm obiectele descrescător după câştigul pe unitatea de greutate (valoare care constituie o măsură a eficienţei transportării obiectelor). Cât timp este posibil (încap în rucsac), selectăm obiectele în întregime. Completăm rucsacul cu un fragment din următorul obiect ce nu a fost selectat.

```
#include <iostream.h>
int o[50]; //ordinea obiectelor
float c[100], g[100], x[100], Gr, GMax;
int n;
int main()
{int i, schimb, aux;
 cout << "n= "; cin >> n;                          //citire
 cout<< "GMax= "; cin >> GMax;
 cout<<"Castigul si greutatea pt. fiecare obiect\n";
 for (i=0; i<n; i++)
     {o[i]=i; cin>>c[i]>>g[i];}
 //ordonez obiectele descrescator dupa castigul unitar
 do
  {schimb=0;
   for (i=0; i<n-1; i++)
       if (c[o[i]]/g[o[i]]<c[o[i+1]]/g[o[i+1]])
          {aux=o[i];o[i]=o[i+1]; o[i+1]=aux; schimb=1;}
  } while (schimb);
 for (i=0, Gr=GMax; i<n && Gr>g[o[i]]; i++)
     {x[o[i]]=1; Gr-=g[o[i]];}
 if (i<n) x[o[i]]= Gr/g[o[i]];
 cout << "Obiectele selectate sunt:\n";
 for (i=0; i<n; i++)
     if (x[i]) cout<<i+1<<' '<<x[i] * 100<<'%'<<endl;
 return 0;}
```

Observație

Am aplicat tot o strategie de tip *Greedy*. Se poate demonstra corectitudinea acestui algoritm în condițiile în care putem încărca în rucsac orice parte a unui obiect. Pentru cazul în care hoțul poate încărca un obiect doar în întregime (varianta discretă a problemei rucsacului), algoritmul *Greedy* nu mai funcționează. De exemplu, pentru `n=3, Gmax=10, g=(8, 6, 4), c=(8, 6, 4)`, algoritmul *Greedy* va obține soluția `x=(1, 0, 0)`, pentru care câștigul obținut în urma transportului este `8`. Soluția optimă este `x=(0, 1, 1)`, pentru care se obține câștigul `10`.

Generare de submulțimi

Fie `n` un număr natural nenul. Să se genereze toate submulțimile mulțimii `{0, 1, ..., n-1}`.

Soluție

Vom reprezenta submulțimile prin vector caracteristic (un vector `S` cu `n` componente `0` sau `1`, având semnificația `S[i]=1`, dacă `i` aparține submulțimii, și `0`, în caz contrar). Problema se reduce astfel la generarea tuturor secvențelor binare de lungime `n`. Pentru a rezolva această problemă vom utiliza un algoritm de tip succesor. Mai exact, un astfel de algoritm funcționează astfel:

Pas 1. Se inițializează soluția cu cea mai mică configurație posibilă (în cazul nostru, inițializăm vectorul `S` cu `0`, ceea ce corespunde mulțimii vide).

Pas 2. Cât timp este posibil (există succesor), se execută:

- se afișează configurația curentă;
- se generează configurația următoare (în cazul nostru, vom considera că elementele vectorului reprezintă cifrele unui număr scris în baza 2, la care vom aduna `1`).

```
#include <fstream.h>
#define NMax 100
int main()
{int n, S[NMax], i, gata=0;
 ofstream fout("subm.out");
 cout<<"n= "; cin>>n;
 for (i=0; i<n; i++) S[i]=0; //initializare
 while (!gata)
     {//afisam configuratia curenta
      for (i=0;i<n;i++) if (S[i]) fout<<i<<' '; fout<<endl;
      //generam configuratia urmatoare prin adunare binara
      for (i=0; i<n && S[i]; i++) S[i]=0;
      if (i==n) gata=1; /* este ultima configuratie  */
          else S[i]=1; }
 fout.close();
 return 0; }
```

Generare elemente produs cartezian

Fie n un număr natural (n>1) și L=(l_1, l_2, ..., l_n) o mulțime de n numere naturale nenule. Să se determine în ordine lexicografică[29] toate elementele produsului cartezian {1, 2, ..., l_1}x{1, 2, ..., l_2}x...x{1, 2, ..., l_n}.

De exemplu, pentru n=3 și L=(2, 3, 2), elementele produsului cartezian sunt: (1,1,1), (1,1,2), (1,2,1), (1,2,2), (1,3,1), (1,3,2), (2,1,1), (2,1,2), (2,2,1), (2,2,2), (2,3,1), (2,3,2). Observați că produsul cartezian are $l_1*l_2*...*l_n$ elemente.

Soluție

Vom reprezenta un element al produsului cartezian ca un vector E cu n elemente, unde E[i]∈{1, 2, ..., l_i}.

Pentru a genera toate elementele produsului cartezian în ordine lexicografică, vom aplica tot un algoritm de tip succesor:

Pas 1. Inițializăm vectorul E cu 1 (cel mai mic element al produsului cartezian, din punct de vedere lexicografic).

Pas 2. Cât timp este posibil (mai există succesor),
- afișăm elementul curent;
- generăm elementul următor; în acest scop, căutăm prima componentă (începând din dreapta către stânga) care poate fi mărită (adică E[i]<L[i]); dacă găsim o astfel de componentă, o mărim și repunem pe 1 toate componentele următoare; dacă nu găsim o astfel de componentă, deducem că generarea s-a încheiat, acesta a fost cel mai mare element din punct de vedere lexicografic.

```
#include <fstream.h>
#define NMax 100
int main()
{int n, L[NMax], E[NMax], i, gata=0;
 ofstream fout("pc.out");
 //citire si initializare
 cout<<"n= "; cin>>n;
 for (i=0; i<n; i++) {cin>>L[i]; E[i]=1;}
 while (!gata)
      {//afisam configuratia curenta
       for (i=0; i<n; i++) fout<<E[i]<<' '; fout<<endl;
       //generam configuratia urmatoare
       for (i=n-1; i>=0 && E[i]==L[i]; i--) E[i]=1;
       if (i<0) /* acesta a fost ultima configuratie */
          gata=1;
          else E[i]++; }
 fout.close(); return 0;}
```

29. Fie x= (x_1, x_2, ..., x_n) și y= (y_1, y_2, ..., y_m). Spunem că x precedă pe y din punct de vedere lexicografic dacă există k astfel încât $x_i=y_i$, pentru orice i<k și $x_k<y_k$ sau k>n.

Expresie

Se dă un şir de n numere naturale nenule $x_1, x_2, \ldots, x_n$ şi un număr natural m.

Scrieţi un program care să verifice dacă valoarea expresiei $\sqrt[m]{x_1 x_2 \ldots x_n}$ este un număr natural. În caz afirmativ, să se afişeze acest număr descompus în factori primi.

Date de intrare

În fişierul `exp.in` se află pe prima linie m, pe linia a doua n, iar pe linia a treia numerele $x_1, x_2, \ldots, x_n$ separate între ele prin câte un spaţiu.

Date de ieşire

În fişierul `exp.out` se va scrie pe prima linie cifra 0, dacă valoarea expresiei nu este un număr natural, respectiv 1, dacă este un număr natural. Dacă valoarea expresiei este un număr natural, pe următoarele linii se vor scrie perechi de forma p e (p este factor prim care apare în descompunere la puterea e≥1). Aceste perechi se vor scrie în ordine crescătoare după primul număr (adică p).

Restricţii

- n – număr natural nenul <5000
- x_i – număr natural nenul <30000, i∈{1, 2, ..., n}

Exemple

exp.in	exp.out	exp.in	exp.out
2 4 32 81 100 19	0	2 4 32 81 100 18	1

(Olimpiada Judeţeană de Informatică, 2004, clasa a IX-a)

Soluţie

Numărul de valori fiind mare, nu putem calcula produsul $x_1*x_2*\ldots*x_n$ deoarece ar depăşi, cu siguranţă, valoarea maximă ce poate fi reprezentată folosind tipurile întregi ale limbajului.

Prin urmare, vom adopta o altă strategie: determinăm descompunerea în factori primi a acestui produs, fără a calcula produsul.

Pentru aceasta, descompunem în factori primi fiecare valoare x_i pe care o citim. Pentru a lucra eficient cu descompunerile în factori primi, vom considera un vector `fp` cu maxim 30000 de componente (30000 fiind valoarea maximă pe care o poate lua oricare număr dintre cele date). Dacă i este număr prim, `fp[i]` va reprezenta multiplicitatea factorului prim i în produsul $x_1*x_2*\ldots*x_n$; dacă i nu este număr prim, `fp[i]` rămâne 0.

Condiția ca expresia din enunțul problemei să fie un număr natural este ca toți factorii primi care intervin în produs să aibă multiplicitățile divizibile cu m (mai exact, toate componentele vectorului fp să fie divizibile cu m).

```
#include <fstream.h>
unsigned int fp[30000];
int main ()
{ifstream fin("exp.in");
 ofstream fout("exp.out");
 int n, m, x, i, d;
 //citire
 fin >> m >> n;
 for (i=0; i<n; i++)
     {
      fin>>x;
      //descompun pe x in factori primi
      for (d=2; x>1; d++)
          while (!(x%d))
                { fp[d]++; x/=d;}
     }
 /* verific daca toti factorii primi au multiplicitatile
    divizibile cu m */
 for (i=2; i<30000 && !(fp[i]%m); i++);
 if (i<30000)
    /* exista cel putin un factor prim pentru care
       multiplicitatea nu este divizibila cu m, deci
       rezultatul expresiei nu este numar natural */
    fout << 0 << '\n';
    else
    {
    fout<<1<<'\n';
    for (i=2;i<30000; i++)
        if (fp[i])
           fout<<i<<' '<<fp[i]/m<<'\n';
    }
 fout.close(); return 0; }
```

Reactivi

Într-un laborator de analize chimice se utilizează N reactivi. Se știe că, pentru a evita accidentele sau deprecierea reactivilor, aceștia trebuie să fie stocați în condiții de mediu speciale. Mai exact, pentru fiecare reactiv x se precizează intervalul $[min_x, max_x]$ în care trebuie să se încadreze temperatura de stocare a acestuia. Reactivii vor fi plasați în frigidere. Orice frigider are un dispozitiv cu ajutorul căruia putem stabili temperatura (constantă) ce va fi în interiorul aceluia (exprimată într-un număr întreg de grade Celsius).

Scrieți un program care să determine numărul minim de frigidere necesare pentru stocarea reactivilor chimici.

Date de intrare

Fișierul de intrare `react.in` conține:
- pe prima linie, numărul natural `N`, care reprezintă numărul de reactivi;
- pe fiecare dintre următoarele `N` linii se află `min max` (două numere întregi separate printr-un spațiu); numerele de pe linia `x+1` reprezintă temperatura minimă, respectiv temperatura maximă de stocare a reactivului `x`.

Date de ieșire

Fișierul de ieșire `react.out` va conține o singură linie pe care este scris numărul minim de frigidere necesar.

Restricții

- $1 \leq N \leq 8000$;
- $-100 \leq min_x \leq max_x \leq 100$ (numere întregi, reprezentând grade Celsius), pentru orice x de la 1 la N;
- un frigider poate conține un număr nelimitat de reactivi.

Exemple

react.in	react.out	react.in	react.out	react.in	react.out
3 -10 10 -2 5 20 50	2	4 2 5 5 7 10 20 30 40	3	5 -10 10 10 12 -20 10 7 10 7 8	2

(Olimpiada Județeană de Informatică, 2004, clasa a IX-a)

Soluție

Intervalele de temperatură pot fi considerate segmente de dreaptă. Problema cere, de fapt, determinarea unui număr minim de puncte, astfel încât orice segment să conțină cel puțin unul dintre punctele determinate.

Vom sorta, în primul rând, intervalele de temperatură crescător după temperatura minimă și descrescător după temperatura maximă.

Deschidem un frigider și plasăm primul reactiv în acest frigider. Pentru frigiderul curent reținem temperatura minimă și temperatura maximă (intervalul de temperatură în care poate fi setat).

Parcurgem succesiv reactivii (în ordine) și pentru fiecare reactiv verificăm dacă el poate fi plasat în frigiderul curent (pentru aceasta, trebuie ca intersecția dintre intervalul de temperatură al frigiderului și intervalul de temperatură al reactivului să fie nevidă). Dacă da, plasăm acest reactiv în frigiderul curent (actualizând corespunzător

intervalul de temperatură al frigiderului). În caz contrar, deschidem un nou frigider (intervalul de temperatură al acestui frigider va fi intervalul reactivului plasat în el).

```
#include <fstream.h>
ifstream fin("react.in");
ofstream fout("react.out");
int N;
int ti[8000], tf[8000];
/*ti[i]=temperatura minima pentru reactivul i
  tf[i]=temperatura maxima pentru reactivul i */
int main ()
{int nf, poz, miny, maxx, max, i, j, icx, icy;
 //citire
 fin>>N;
 for (i=0;i<N;i++) fin>>ti[i]>>tf[i];
 fin.close();
 //sortare prin selectia maximului
 for (i=N-1; i>0; i--)
     {for (poz=i, j=0; j<i; j++)
          if (ti[j]>ti[poz] ||
              ti[j]==ti[poz] && tf[j]<tf[poz]) poz=j;
      max=ti[poz];ti[poz]=ti[i]; ti[i]=max;
      max=tf[poz];tf[poz]=tf[i]; tf[i]=max;  }
/* deschidem un frigider si plasam primul reactiv
icx=temperatura minima de functionare a frigiderului curent
icy=temperatura maxima de functionare a frigiderului curent
nf=numarul de frigidere deschise */
 nf=1; icx=ti[0];icy=tf[0];
//parcurg ceilalti reactivi in ordine
 for (i=1; i<N; i++)
     {
/*   verificam   daca   intervalul   curent   icx,   icy   se
intersecteaza
cu intervalul de temperatura al reactivului i */
      miny=icy;if (miny>tf[i]) miny=tf[i];
      maxx=icx;if (maxx<ti[i]) maxx=ti[i];
      if (maxx <= miny)
         //actualizez intervalul de temperatura
         {icx=maxx;icy=miny;}
      else //deschid un nou frigider
         {nf++;
          icx=ti[i];icy=tf[i]; }
     }
 fout<<nf<<'\n';
 fout.close(); return 0;}
```

Furnici

Una dintre cele mai importante funcții din colonia de furnici este cea de ofițer de circulație. Motivul este că în mușuroiul de furnici există un tunel foarte îngust, astfel încât prin tunel nu pot să treacă două furnici una pe lângă alta. Din fericire, în tunel există niște locuri speciale, unde furnicile pot sta și pot aștepta până trec alte furnici. Furnicile pot trece una pe lângă alta fără nici o problemă la capetele tunelului. Ofițerul de circulație cunoaște lungimea tunelului, precum și pozițiile locurilor speciale, în care furnicile pot staționa. În fiecare dimineață, ofițerul de serviciu primește o listă cu toate sosirile în tunel. Misiunea sa este de a organiza traficul astfel încât toate furnicile să poată traversa tunelul. Bineînțeles că acest lucru poate fi realizat în multe moduri, mai rapid sau mai lent. Deoarece atunci când ultima furnica va ieși din tunel ofițerul de serviciu își poate încheia programul, interesul său este de a organiza traficul astfel încât acest lucru să se întâmple cât mai repede. Orice furnică se deplasează cu `1` cm/s. În locurile speciale pot aștepta oricâte furnici, iar dimensiunile unui astfel de loc pot fi neglijate (poate fi considerat punctiform).

Cerință

Scrieți un program care să determine timpul minim necesar pentru ca toate furnicile să traverseze tunelul.

Date de intrare

Fișierul de intrare `furnici.in` conține pe prima linie două numere naturale `D` și `U`, separate printr-un spațiu; `D` reprezintă lungimea tunelului exprimată în cm, iar `U` reprezintă numărul de locuri speciale din tunel. Fiecare dintre următoarele `U` linii conține poziția unui loc special din tunel; poziția este dată prin distanța față de capătul din stânga al tunelului (exprimată în cm); pozițiile sunt date în ordinea de la stânga la dreapta. Pe următoarea linie se află un număr natural `L` care reprezintă numărul de furnici care intră prin capătul din stânga al tunelului. Fiecare dintre următoarele `L` linii conține timpul de sosire a unei furnici la capătul din stânga al tunelului. Pe următoarea linie se află un număr natural `R` care reprezintă numărul de furnici ce intră prin capătul din dreapta al tunelului. Fiecare dintre următoarele `R` linii conține timpul de sosire a unei furnici la capătul din dreapta al tunelului.

Date de ieșire

Fișierul de ieșire `furnici.out` conține o singură linie pe care se află timpul minim necesar pentru ca toate furnicile să traverseze tunelul, exprimat în secunde.

Restricții

- $1 \leq D, U, L, R \leq 10000$;
- $D > U$
- timpii de sosire sunt exprimați în secunde și numere naturale ≤ 2000000.

Exemple

furnici.in	furnici.out	furnici.in	furnici.out
5 1 2 1 3 1 2	8	10 2 4 6 2 0 4 1 0	14

(Concursul „Urmaşii lui Moisil", Iaşi, 2004)

Soluţie

Ideea de rezolvare a problemei se bazează pe următoarea observaţie: „Oricât de multe furnici ar fi, timpul maxim în care *toate* furnicile traversează tunelul depinde doar de ultimele două furnici – cea care intră ultima prin capătul din stânga şi cea ce intră ultima prin capătul din dreapta al tunelului, fiindcă atunci toate celelalte furnici au trecut deja". Deci, pentru a determina timpul minim în care toate furnicile trec prin tunel, trebuie determinat timpul minim în care cele două ultime furnici trec prin tunel. Acesta se adaugă apoi la ultimul timp, cel la care intră în tunel ultima furnică (fie în stânga, fie în dreapta) – de fapt, determinarea se face direct, în timpul calculului.

```
#include <fstream.h>
#define INFILE          "furnici.in"
#define OUTFILE         "furnici.out"
#define MaxF            10000 + 1
#define MaxLoc          10000 + 1
#define Infinit         200000000
long int l, m, n, nr_loc, maxL, maxR, rez=Infinit;
long int loc[MaxLoc];
int main(void)
{ifstream fin(INFILE);
 long int i, timp, st, dr, poz;
 fin>>l>>nr_loc; //lungime tunel si numar locuri speciale
 for (i=1; i<=nr_loc; i++) fin>>loc[i];
 fin>>m;  //cate furnici intra prin stanga
 maxL=0; //ultimul moment la care intra o furnica in stanga
 maxR=0;//ultimul moment la care intra o furnica in dreapta
 for (i=1; i<=m; i++)
     //retin timpul cand ultima furnica intra prin stanga
    {//restul nu intereseaza
      fin>>timp;
      if (timp>maxL) maxL=timp;     }
 fin>>n;           //cate furnici intra prin dreapta
 for (i=1; i<=n; i++)
```

```
    //retin timpul cand ultima furnica intra prin dreapta
   {//restul nu intereseaza
     fin>>timp; if (timp>maxR) maxR=timp;    }
 loc[0]=0;//primul loc special este 0 (intrarea din stanga)
 //ultimul loc special este intrarea din dreapta
 loc[nr_loc+1]=l;
 fin.close ();
/*nu ne intereseaza decat ultimele furnici care intra in
tunel prin stanga si prin dreapta, deci pentru aceste doua
furnici se face calculul timpului */
 for (i=0; i<=nr_loc+1; i++)
    {/* pentru fiecare dintre locurile speciale determin
     timpul maxim cand ajunge ultima furnica din stanga
     in acest loc */
     st=maxL+loc[i];
     /* si timpul maxim cand ajunge ultima furnica din
        dreapta in acest loc */
     dr=maxR-loc[i]+l;
     /* in acest loc se ajunge dupa poz secunde, poz fiind
        maximul dintre furnica stanga si furnica dreapta */
      if (st>dr) poz=st;
         else    poz=dr;
      /*  se adauga eventualul timp de asteptare pana
      ajunge cealalta furnica */
      if (loc[i]>l-loc[i]) poz=poz+loc[i];
          else             poz=poz+l-loc[i];
      /* se retine minimul pana in acest moment */
      if (poz<rez) rez=poz;
    }
 //afisare
 ofstream fout(OUTFILE);
 fout<<rez<<endl;
 fout.close();  return 0; }
```

Apariţii cifră

Se consideră numerele naturale de la 0 la N (N≤999999999) şi o cifră nenulă K. Scrieţi un program care să calculeze câte dintre aceste numere conţin în scrierea lor cel puţin o cifră K. De exemplu, pentru N=13 şi K=1, programul va afişa 5.

Soluţie

O primă idee este de a verifica pentru fiecare număr natural din intervalul [0, N] dacă el conţine sau nu în scrierea sa cifra K. Deoarece valoarea lui N este foarte mare, această soluţie este practic inutilizabilă.

O altă idee ar fi de a determina c[i]= numărul de numere naturale din intervalul [0, 10^i-1] care conţin cifra K.

c[1]= numărul de numere naturale din intervalul [0, 9] care conţin cifra K;

c[2]= numărul de numere naturale din intervalul [0, 99] care conţin cifra K;
c[3]= numărul de numere naturale din intervalul [0, 999] care conţin cifra K;
(...)
c[10]= numărul de numere naturale din intervalul [0, 999999999] care conţin cifra K.

Observăm că, indiferent de valoarea cifrei nenule K, vectorul c va avea aceleaşi valori şi că există o relaţie de recurenţă între c[i] şi c[i-1]:
$c[i]=9*c[i-1]+10^{i-1}$.

Dacă prima cifră este K, atunci toate cele 10^{i-1} numere corespund. Dacă prima cifră este diferită de K, atunci numărul de numere este c[i-1]*9.

Pentru a calcula c[i], trebuie să calculăm în prealabil puterile lui 10 (în vectorul P10).

Pentru a calcula numărul total de cifre (NT) vom determina două puteri consecutive ale lui 10 astfel încât $10^p \le N < 10^{p+1}$. Pentru aceasta vom calcula puterile lui 10, în vectorul P10 şi îl vom parcurge de la sfârşit spre început, până determinăm p. Prima cifră a lui N este N/P10[p]. Sunt trei cazuri posibile (prima cifră a lui N este egală cu K , mai mică sau mai mare decât K). Adunăm la NT numărul de numere corespunzător primei cifre a lui N, eliminăm prima cifră din N şi reluăm procedeul.

```
#include <stdio.h>
#define CIFREMAX 10
long int N, K, NT = 0;
long int C[CIFREMAX], P10[CIFREMAX];
int main(void)
{int i, p, pcifra;
 scanf("%d %d",&N,&K);
 C[0]=0; P10[0]=1;
 for (i=1; i<CIFREMAX; i++)
     {C[i]=9*C[i-1]+P10[i-1]; P10[i]=P10[i-1]*10;}
 for (p = CIFREMAX-1; p+1; p--)
     if (N>=P10[p])
         {pcifra=N/P10[p]; //determin prima cifra a lui N
          if (pcifra>K)
             {NT += P10[p]; NT += (pcifra-1)*C[p];  }
          if (pcifra==K)
            {NT+= pcifra*C[p]; NT+= N%P10[p]+1;
             break;                                  }
          if (pcifra<K)  NT+=pcifra*C[p];
          N%=P10[p]; } //eliminam prima cifra a lui N
 printf("%d\n",NT); return 0; }
```

Examen de capacitate

Se citesc de la tastatură numărul de elevi care au participat la examenul de capacitate ($n \le 1000$) şi pentru fiecare elev se citesc numele, prenumele şi cele trei note obţinute la matematică, limba română şi istorie/geografie. Să se calculeze pentru

fiecare elev media obținută la examen și să se afișeze numele și prenumele elevilor în ordine alfabetică și mediile obținute de aceștia.

Soluție

Vom reține informațiile despre elevii care au participat la examenul de capacitate într-un vector `Ex`, fiecare componentă a vectorului fiind o structură cu următoarele câmpuri: `np` (numele și prenumele elevului), `r` (nota la română), `m` (nota la matematică), `ig` (nota la istorie sau geografie) și `mg` (media generală).

```
#include <iostream.h>
#include <string.h>
int main ()
{ struct Elev {char np[40];
               float r, m, ig, mg;} Ex[1000], aux;
  int n, i, schimb;
  cout <<"n="; cin >> n;
  cout << "Introduceti informatiile despre elevi\n";
  for (i=0; i<n; i++)
      {cout << "Numele si prenumele: ";
       cin.get(); cin.getline(Ex[i].np,40);
       cout <<"Notele (R/M/IG): ";
       cin>>Ex[i].r>>Ex[i].m>>Ex[i].ig;}
  //calculam mediile generale
  for (i=0; i<n; i++)
       Ex[i].mg=(Ex[i].r+Ex[i].m+Ex[i].ig)/3;
  //ordonez elevii alfabetic
   do {schimb=0;
       for (i=0; i<n-1; i++)
           if (strcmp(Ex[i].np, Ex[i+1].np)>0)
              { aux=Ex[i]; Ex[i]=Ex[i+1]; Ex[i+1]=aux;
                schimb=1;}
      } while (schimb);
  //afisez
  for (i=0; i<n; i++) cout<<Ex[i].np<<' '<<Ex[i].mg<<endl;
  return 0;}
```

Observații

1. Pentru ordonarea alfabetică a trebuit să comparăm numele elevilor. Deoarece acestea sunt șiruri de caractere, am utilizat funcția `strcmp()`.
2. Înainte de a citi numele unui elev, am apelat funcția `get()` pentru a extrage caracterul `'\n'` tastat după introducerea valorilor numerice precedente.

Aparițiile unui cuvânt

Se citește de la tastatură un cuvânt de maximum `20` de litere și, începând cu linia următoare, un text terminat cu `'\n'`, constituit din maximum `1000` de caractere. Scrieți un program care să numere de câte ori apare cuvântul în text.

Soluţie

Pentru a căuta cuvântul în text, vom utiliza funcţia `strstr()`. Deoarece această funcţie nu determină decât prima apariţie, o vom apela repetat (după fiecare căutare cu succes, vom reapela `strstr()`, dar specificând ca adresă de început al şirului în care se caută nu începutul textului, ci adresa caracterului care urmează după apariţia găsită).

```
#include <iostream.h>
#include <string.h>
int main ()
{char c[20], t[1000], *p; int nr=0;
 cout << "Cuvantul cautat: "; cin.getline(c, 20);
 cout << "Introduceti textul \n"; cin.getline(t,1000);
 p=t;                          //cu pointerul p parcurg textul
 do {
     p=strstr(p,c);                          //caut cuvantul c
     if (p)                                 //am gasit o aparitie
        {nr++;                                        // o numar
         p++;}                                //caut mai departe
     } while (p);
 cout << "Nr de aparitii: "<<nr<<endl; return 0;}
```

Propoziţie

Se citeşte de la tastatură un text scris pe mai multe linii, constituit din propoziţii terminate cu `'.'`, alcătuite din maximum `500` de caractere. Sfârşitul textului este marcat de întâlnirea secvenţei `"#."`. Să se determine cea mai lungă propoziţie din text, în cazul în care:

a. Lungimea unei propoziţii este exprimată în număr de caractere.
b. Lungimea unei propoziţii este exprimată în număr de cuvinte.

Două cuvinte consecutive pot fi separate prin unul sau mai multe caractere din mulţimea `{' ',';',',',':','-',')','('}`.

Soluţie

Se citeşte textul propoziţie cu propoziţie. Pentru fiecare propoziţie citită se determină lungimea şi apoi se compară cu lungimea maximă. Dacă lungimea propoziţiei este mai mare decât lungimea maximă, reţinem această propoziţie şi lungimea sa ca fiind maximă.

```
#include <iostream.h>
#include <string.h>
int main ()
{char p[500], pmax[500];
 int lg, lgmax=0, gata=0;
 do
  {cin.getline(p,500,'.');              //citesc o propozitie
```

```
    if (!strcmp(p,"#")) gata=1;        //s-a terminat textul
        else
        {//calculez lungimea propozitiei citite
         lg=strlen(p);
         if (lg>lgmax) //compar lungimea cu lungimea maxima
            { lgmax=lg; strcpy(pmax,p);}}
        }
  while (!gata);
  cout <<"lgmax="<<lgmax<<" pmax="<<pmax; return 0;}
```

În varianta b se modifică numai modalitatea de calcul al lungimii unei propoziții. Pentru aceasta se extrag succesiv cuvintele din propoziție, utilizând funcția `strtok()` și se numără:

```
char copie[500], sep[]=" ;,:-)(", *c;
 lg=0;
 strcpy(copie,p);                          //copiez propozitia
 c=strtok(copie, sep);
 while (c)                                 //am extras un cuvant
       {lg++;                                        //il numar
        c=strtok(NULL, sep);}  //incerc sa extrag urmatorul
```

Observație

A fost necesară copierea propoziției și extragerea cuvintelor din copie, deoarece funcția `strtok()` distruge șirul din care extrage unitățile lexicale.

4.12. *Probleme propuse*

1. Care dintre următoarele secvențe de instrucțiuni determină în variabila reală `min` cel mai mic element din vectorul `a` de `n` numere reale?

(variantă Bacalaureat, 2000)

a.
```
min=0;
for (i=1; i<n; i++)
     if (min<a[i]) min=a[i];
```

b.
```
min=a[0];
for(i=1;i<n;i++)
     if (min<a[i]) min=a[i];
```

c.
```
min=a[0];
for (i=1; i<n; i++)
     if (min>a[i]) min=a[i];
```

d.
```
min=a[n-1];
for (i=1; i<n; i++)
     if(min>a[i-1])min=a[i-1];
```

2. Care dintre următoarele declarații reprezintă un tablou cu `2` linii și `4` coloane cu elemente întregi?

a.
```
longint a[2][4];
```

b.
```
unsigned int a[2][4];
```

c.
```
int a[4][2];
```

d.
```
long a[2][4];
```

3. Fie `A` o matrice cu `n` linii (numerotate de la `0` la `n-1`) şi `m` coloane (numerotate de la `0` la `m-1`) cu elemente întregi. Ce proprietate a elementelor matricei `A` testează următoarea secvenţă de instrucţiuni?

```
for (ok=1, i=0; i<m && ok; i++)
     {for  (ok=j=0; j<n && !ok; j++)
            if (a[j][i]) ok=1;     }
if (ok) cout<<"da"; else cout<<"nu";
```

a. Toate elementele matricei `A` sunt nenule.
b. Există o coloană a matricei `A` care conţine cel puţin un element nenul.
c. Există o linie a matricei `A` care are toate elementele nenule.
d. Toate coloanele matricei `A` conţin cel puţin un element nenul.
e. Toate coloanele matricei `A` au toate elementele nenule.
f. Există un element nenul în matricea `A`.

4. Fie `A` o matrice cu `n` linii (numerotate de la `0` la `n-1`) şi `m` coloane (numerotate de la `0` la `m-1`) cu elemente întregi. Ce proprietate a elementelor matricei `A` testează următoarea secvenţă de instrucţiuni?

```
for (ok=1,i=0; i<n && ok; i++)
    for  (j=0, k=m-1; j<k && ok; j++, k--)
          if (a[i][j]!=a[i][k]) ok=0;
if (ok) cout<<"da";  else cout<<"nu";
```

a. Orice linie a matricei `A` are elementele egale.
b. Oricare două elemente ale matricei `A` sunt diferite.
c. Există două elemente diferite în matricea `A`.
d. Oricare două elemente de pe aceeaşi linie a matricei `A` sunt diferite.
e. Oricare două elemente de pe aceeaşi linie a matricei `A`, simetrice faţă de mijlocul liniei, sunt egale.
f. Există o linie a matricei `A` în care elementele situate la egală distanţă faţă de capetele liniei sunt egale.

5. Fie `A` un vector cu `n` elemente întregi. Care dintre următoarele secvenţe de instrucţiuni inversează ordinea elementelor plasate în vector de la poziţia `st` până la poziţia `dr` (`st<dr`)?

a.
```
for (i=st,j=dr;i<j;i++,j--)
    {aux=A[st];
     A[st]=A[dr];
     A[dr]=aux;}
```

b.
```
for (; st<dr; st++, dr--)
      {aux=A[st];
       A[st]=A[dr];
       A[dr]=aux;}
```

c.
```
for(i=st;i<=(dr-st+1)/2;i++)
   {aux=A[i];
    A[i]=A[dr-st+i];
    A[dr-st+i]=aux;}
```

d.
```
for (i=st;i<=(st+dr)/2;i++)
       {aux=A[i];
        A[i]=A[dr+st-i];
        A[dr+st-i]=aux;}
```

6. Ce va afişa pe ecran următorul program?

```
#include <iostream.h>
void main()
{int n=10, i;
 int a[]={1,2,3,4,5,6,7,8,9,10};
 for (i=n/2; i; i--) a[i+n/2-1]=a[i];
 for (i=0; i<n; i++) cout<<a[i];}
```

a. Nu se va afişa nimic, deoarece se obţine eroare la compilare.
b. 1234562345 **d.** 1234554321 **f.** 1234523456
c. 1234512345 **e.** 1234567891

7. Se consideră următoarea secvenţă:

```
int a[4][4] = {1,2,3,4,5,6,7,8,9,1,2,3,4,5,6,7},n=4,x=0,i;
for (i=0; i<n-1; i++) x+=a[i][n-i-2];
```

Care va fi valoarea variabilei x după execuţia acestei secvenţe?

a. 15 **b.** 16 **c.** 18 **d.** 20
e. secvenţa nu se execută, deoarece se obţine eroare la compilare.

8. Unde ar trebui inserată instrucţiunea `i++;` în secvenţa următoare astfel încât în urma execuţiei secvenţei să fie eliminate toate elementele care erau iniţial pe poziţii pare în vectorul a cu n elemente întregi, plasate în vector pe poziţiile 1, 2, ..., n?

```
i=1;
while (i<=n)
    {for (j=i+1; j<=n; j++) a[j-1]=a[j];
     n--;    }
```

a. Nicăieri, deoarece nu este necesar.
b. O dată înainte şi o dată după instrucţiunea `for`.
c. O dată înainte de instrucţiunea `for`.
d. O dată după instrucţiunea `for`.
e. De două ori după instrucţiunea `for`.
f. De două ori înainte de instrucţiunea `for`.

9. Ce va afişa pe ecran următorul program, dacă se citesc de la tastatură valorile:
4 0 4 2 3 1 1 2 4 3 2 1 5

```
#include <iostream.h>
int a[5][5];
void main()
{int nr, i,x,y,z;
 cin>>nr;
 for (i=0; i<nr; i++) {cin>>x>>y>>z; a[x][y]=z;}
 for (nr=i=0; i<5; i++) nr+=a[i][4-i];
 cout<<nr; }
```

a. 11 **b.** 5 **c.** 1 **d.** 3 **e.** 2 **f.** 8

10. Ce va afişa pe ecran următoarea secvenţă de instrucţiuni?

```
int *p, *q;
int a=3;
float b=7.5;
*p=a; *q=b;
cout<<*p<<' '<<*q;
```

a. 3 7 **b.** 7 3 **c.** 3 7.5 **d.** 7.5 3

e. Nimic, se produce eroare la execuţia programului.

11. Să considerăm următoarele declaraţii de variabile:

```
char s[]="Era Vasile om frumos", *q=strstr(s,"Vasile");
```

a. Care este efectul următoarei atribuiri?

```
*(q+1)='y';
```

b. Dar al atribuirii următoare?

```
*q+1='x';
```

12. Care este efectul următoarei secvenţe?

```
char s[]="Era Vasile om frumos", *q=strchr(s,'a');
char *p=strchr(s,'f');
while (++q < p++)
      {*p='x';
       *q='y';
       p-=2; q+=2;  }
puts(s);
```

13. Care este efectul următoarei secvenţe?

```
char s[]="abracadabra", *q=s, *p=s+strlen(s)-1;
while (p!=s && ++*q)
      {*p=*q;
       p--; q++;  }
puts(s);
```

14. Fie `s` un şir de `n` caractere (`n` par, `n≤100`). Care dintre următoarele secvenţe inversează prima jumătate a şirului `s` cu cea de-a doua jumătate (de exemplu, şirul „`mamatata`" ar deveni „`tatamama`").

a.

```
int i, n=strlen(s), m=n/2-1;
char v;
for (i=0; i<m; i++)
{v=s[i];s[i]=s[m+i];s[m+i]=v;}
```

b.

```
int i, n=strlen(s), m=n/2;
char v;
for (i=0; i<m; i++)
{v=s[i];s[i]=s[m+i];s[m+i]=v;}
```

d.

```
char *v;
int n=strlen(s), m=n/2;
strcat(v,s+m); s[m]=NULL;
strcat(v,s); strcpy(s,v);
```

e.

```
int n=strlen(s),m=n/2;
strncpy (s+n,s,m);
*(s+n+m)=0;
strcpy (s,s+m);
```

c.
```
char v[101];
int n=strlen(s),m=n/2;
strcat(v,s+m); s[m]='\n';
strcat(v,s); strcpy(s,v);
```

f.
```
char v[201]="";
int n=strlen(s),m=n/2;
strcat(v,s+m); s[m]='\0';
strcat(v,s); strcpy(s,v);
```

15. Se consideră următorul program:

```
#include <iostream.h>
char s[100];
int uz[26], i;
int main()
{cin>>s;
 for (i=0; s[i]; i++)
    uz[s[i]-'a']++;
 for (i=0; i<26; i++)
      if (uz[i]) cout<<uz[i];
 return 0;}
```

Ce valoare ar putea fi introdusă de la tastatură pentru variabila `s` pentru a obține pe ecran `11211`?

a. `Cabana`
b. Nici o valoare, pentru că programul este greșit din punct de vedere sintactic.
c. `viitor`
d. `osmoza`
e. `activi`
f. `baaaaabaaaaaasm`

16. Scrieți un program care citește de la tastatură cele `10` numere reale ce compun vectorul `a` și apoi cele `8` numere reale ce compun vectorul `b` și afișează pe ecran câte dintre componentele vectorului `a` sunt strict mai mici decât toate componentele vectorului `b`. De exemplu, dacă `a=(4, 8, 1, 9, 5, 11, 3, 43, 6, 20)` și `b=(9, 9, 6, 9, 9, 8, 6, 9)`, atunci veți afișa `4`, deoarece valorile `4`, `1`, `5` și `3` sunt mai mici decât toate elementele lui `b`.

(variantă Bacalaureat, 2000)

17. Scrieți un program care citește de la tastatură cele `20` de componente reale ale unui vector `a` și afișează pe ecran câte dintre valorile citite sunt mai mici decât media aritmetică a componentelor vectorului. De exemplu, dacă valorile citite sunt `3.2, 9, -2, 0, 4, -4, -0.5, 7, 1, 0.3, 14, 0, -7, 2, 2, -1, 3, 6, 0, 1`, se va afișa pe ecran valoarea `11`.

(variantă Bacalaureat, 2001)

18. Fie `a` un vector cu `n` (`n≤50`) componente de tip `int`.

a. Verificați dacă toate elementele vectorului `a` sunt numere prime.
b. Verificați dacă există în vectorul `a` un palindrom (număr care, citit de la stânga la dreapta și de la dreapta la stânga, este același).

c. Eliminați din vector toate elementele impare.
d. Inserați în vector după fiecare element egal cu `0` un element cu valoarea `1`.
e. Determinați cea mai lungă secvență formată din elemente egale existentă în vector. Afișați lungimea secvenței, poziția de început și valoarea care se repetă.
f. Determinați un element din vector care se repetă de cele mai multe ori.

19. Fie `a` o matrice cu `n` linii și `m` coloane (`n,m≤100`) cu componente de tip `int`.
a. Verificați dacă există o coloană în matrice cu toate elementele nule.
b. Calculați și afișați produsul elementelor nenule de pe fiecare linie a matricei.
c. Numărați câte elemente din matrice au ca vecini numai numere pare (două elemente sunt numite vecine dacă sunt adiacente pe verticală sau orizontală).

20. Fie `a` o matrice pătratică cu `n` linii (`n≤100`) cu componente de tip `int`.
a. Verificați dacă matricea este inferior triunghiulară (toate elementele de deasupra diagonalei principale sunt nule).
b. Verificați dacă matricea este simetrică față de diagonala principală.
c. Calculați suma elementelor impare de sub diagonala secundară.
d. Determinați sumele elementelor de pe chenarele concentrice ale matricei.
De exemplu, dacă `n=5` și matricea este:

1	2	9	0	3
9	0	7	6	4
8	7	1	2	0
3	6	2	0	9
1	3	7	0	0

se formează trei chenare concentrice, sumele (dinspre exterior spre interior) fiind: `59, 30, 1`.

21. Pe prima linie a fișierului text `BAC2000.TXT` se găsește o succesiune de cel puțin două și cel mult `2000` de caractere ce pot fi doar litere mari și spații. Scrieți un program care citește de la tastatură un număr natural `k` (`0<k<2000`) și stabilește dacă există în fișier vreo literă care apare de exact `k` ori. Programul afișează pe ecran mesajul `Da` în cazul în care există cel puțin o literă cu proprietatea menționată și mesajul `Nu`, în caz contrar. De exemplu, dacă fișierul `BAC2000.TXT` are următorul conținut:
`EXAMEN DE BACALAUREAT LA INFORMATICA`
iar de la tastatură se citește numărul `8`, atunci, deoarece litera `A` apare de exact `8` ori, se va afișa pe ecran mesajul `Da`.

(variantă Bacalaureat, 2000)

22. *Melci*

Se consideră n stâlpi (numerotați de la 1 la n, n<20) de înălțimi h_1, h_2, ..., h_n metri. La baza fiecărui stâlp se află câte un melc codificat prin numărul stâlpului. Fiecare melc i urcă ziua p_i metri și coboară noaptea q_i metri ($p_i \geq q_i$). Scrieți un program care să afișeze melcii în ordinea în care ating vârfurile stâlpilor.

De exemplu, să considerăm trei stâlpi cu înălțimile 2, 4, 5. Melcul aflat la baza stâlpului 1 urcă 1 m pe zi și coboară 0 m pe noapte. Melcul aflat la baza stâlpului 2 urcă 4 m pe zi și coboară 1 m pe noapte. Melcul aflat la baza stâlpului 3 urcă 1 m pe zi și coboară 0 m pe noapte.

Melcul 1 ajunge în vârful stâlpului după două zile. Melcul 2 ajunge în vârful stâlpului după o zi. Melcul 3 ajunge în vârful stâlpului după cinci zile. Deci ordinea în care melcii ajung în vârf este 2 1 3.

(Concurs PACO, 1997)

23. *Generare tablou*

Scrieți un program care construiește în memorie un tablou T cu n linii și n coloane, cu elemente numere naturale, astfel încât pe ultima coloană și pe ultima linie a tabloului să se afle numai elemente egale cu 1, iar oricare alt element al tabloului să fie suma dintre elementul aflat imediat sub el și elementul aflat imediat în dreapta lui.

Valoarea lui n (număr natural, n<20) se citește de la tastatură. Tabloul se va afișa pe ecran în mod standard: de la prima la ultima linie, elementele unei linii fiind scrise de la stânga la dreapta, cu spații între elementele fiecărei linii.

De exemplu, pentru n=4, se va afișa tabloul următor:

```
20   10   4    1
10    6   3    1
 4    3   2    1
 1    1   1    1
```

Se observă că fiecare element al tabloului (în afara elementelor 1) respectă cerința: 20=10+10, 10=6+4, 4=3+1, 10=4+6, 6=3+3, 3=2+1, 4=1+3, 3=1+2 și 2=1+1.

(variantă Bacalaureat, 2002)

24. *Generare tablou*

Scrieți un program care construiește în memorie un tablou T cu n linii și n coloane, cu elementele numere naturale, astfel încât pe diagonala principală să existe numai elemente egale cu 1, elementele de pe cele două „semidiagonale" paralele cu diagonala principală și alăturate diagonalei principale să fie toate egale cu 2, elementele de pe următoarele două „semidiagonale" să fie egale cu 3 etc. Valoarea lui n (număr natural, n<20) se citește de la tastatură.

Tabloul se va afișa pe ecran cu formatul sugerat în exemplul următor. Pentru n=4, se va afișa tabloul:

```
1 2 3 4
2 1 2 3
3 2 1 2
4 3 2 1
```

(variantă Bacalaureat, 2002)

25. *Sumă şi produs de polinoame*

Fie `P` un polinom de grad `n` şi `Q` un polinom de grad `m` (`n,m<100`) cu coeficienţi reali. Scrieţi un program care calculează suma şi produsul polinoamelor.

26. *Problema instructorului de schi*

Un instructor de schi are la dispoziţie `n` perechi de schiuri (`n≤50`) pe care trebuie să le distribuie la `n` elevi începători. Scrieţi un program care să distribuie cele `n` perechi de schiuri astfel încât suma diferenţelor absolute dintre înălţimea elevului şi lungimea schiurilor atribuite să fie minimă.

27. *Mulţimi*

Se citesc de la tastatură două mulţimi `A` şi `B` cu `n`, respectiv `m` elemente numere naturale mai mici decât `1000`. Să se determine intersecţia şi reuniunea celor două mulţimi. Implementaţi mai întâi o mulţime ca un vector în care memoraţi elementele mulţimii, apoi prin vector caracteristic. Realizaţi o comparaţie între eficienţa implementării celor două operaţii cu cele două reprezentări.

28. *Sumă*

Se consideră un număr natural `n` nenul. Să se elaboreze un program care să afişeze şi să contorizeze toate modalităţile de a scrie acest număr ca sumă de minim două numere întregi consecutive. De exemplu, pentru `n=6` veţi obţine `3` modalităţi: `1+2+3`, `0+1+2+3`, `-5-4-3-2-1+0+1+2+3+4+5+6`.

Fişierul de ieşire `sume.out` va conţine pe prima linie numărul `k` de modalităţi de a scrie numărul ca sumă de numere întregi consecutive, iar pe următoarele `k` linii câte două numere întregi `n s`, separate printr-un spaţiu, reprezentând, respectiv, numărul de termeni şi primul termen din suma corespunzătoare liniei.

29. *Zig-zag*

Din fişierul `ZigZag.in` se citesc `N` numere întregi (`N≤1000`). Afişaţi pe prima linie a fişierului `ZigZag.out` cel mai lung MZigZag care se poate construi din numerele citite. Numim MZigZag o secvenţă de numere a_1, a_2, ..., a_m astfel încât $a_1 \le a_2 \ge a_3 \le a_4 \ge a_5 \le ... \le a_{m-1} \ge a_m$. De exemplu:

ZigZag.in	ZigZag.out
7 7 5 0 1 4 9 3	0 9 3 7 1 5 4

30. *Relații*

Profesorul X a studiat structura relațiilor dintre elevii săi. Pentru a reprezenta această structură, profesorul a numerotat elevii de la 1 la n (n≤100) și a construit o matrice pătratică cu n linii, astfel: a[i][j]=1, dacă elevul i îl agreează pe elevul j, și 0, în caz contrar. În plus, a considerat că fiecare elev se agreează pe el însuși. Această matrice a memorat-o în fișierul Relatii.in.

a. Determinați și afișați pe ecran toate perechile distincte de elevi care se agreează reciproc.

b. Afișați elevii care nu sunt agreați de nimeni.

c. Afișați elevii care nu agreează pe nimeni.

d. Afișați elevii în ordinea descrescătoare a popularității lor. Popularitatea unui elev este egală cu numărul de elevi care îl agreează.

31. Se citește de la tastatură o propoziție terminată cu '.', constituită din maximum 200 de caractere. Scrieți un program care înlocuiește toate literele mici din propoziție cu litere mari, fără a utiliza funcții standard.

32. Fie c un cuvânt și p o propoziție. Să se cerceteze dacă propoziția p conține, în ordine, literele cuvântului c. De exemplu, literele cuvântului Multe se găsesc, în ordine, în propoziția Muntele te cheama.

33. *Agendă telefonică*

Să se scrie un program care citește de la tastatură un număr natural n (n≤200), apoi numele și numărul de telefon ale n persoane. Programul va crea o agendă telefonică în care va reține informațiile citite în ordinea alfabetică a numelor persoanelor. După crearea agendei telefonice, programul va citi de la tastatură numele unei persoane, apoi va căuta persoana în agendă. Dacă persoana va fi găsită, se va afișa pe ecran numărul ei de telefon, altfel se va afișa un mesaj de eroare.

34. Se citește de la tastatură un șir c de maximum 20 de litere și, începând cu linia următoare, un text terminat cu '\n', din maximum 1000 de caractere. Scrieți un program care să șteargă din text toate aparițiile disjuncte ale șirului c.

35. *Multiplu*

Fie n un număr natural (n≤1000). Determinați un multiplu al lui n a cărui scriere în baza 10 conține numai cifrele 1 și 0.

36. *Planificarea optimală a lucrărilor*

Să se planifice optimal (astfel încât penalizarea totală să fie minimă) executarea a n ($n \in \mathbf{N}^*$) lucrări, cunoscând pentru fiecare lucrare i termenul de predare t_i și penalizarea p_i care se plătește în cazul în care lucrarea nu este finalizată la termen. Se știe că pentru execuția unei lucrări este necesară o unitate de timp, că nu se pot executa două lucrări în același timp și toate lucrările trebuie să fie executate. Planificarea constă în specificarea ordinii în care trebuie să fie executate lucrările.

37. *Bac*

Orașul A este situat pe un mal al Dunării, iar orașul B pe celălalt mal. Legătura dintre orașele A și B se poate realiza numai cu bacul. Compania X a primit autorizație de transport fluvial, cu condiția să asigure și legătura cu bacul între orașele A și B. Primăriile celor două orașe au stabilit de comun acord programul sosiri-plecări pe care trebuie să îl respecte compania X.

Fiind un tânăr programator dornic de afirmare, vreți să determinați numărul minim de bacuri necesare companiei, astfel încât programul să poată fi respectat.

Date de intrare

Prima linie a fișierului de intrare `bac.in` conține două numere naturale `K` și `L`, separate printr-un singur spațiu (`K` – durata traversării, `L` – durata de îmbarcare, egală cu durata de debarcare, exprimată în minute).

Următoarea linie conține un număr natural `A` care reprezintă numărul de plecări din orașul `A`. Fiecare dintre următoarele `A` linii conține timpul unei plecări din orașul `A`. Următoarea linie conține un număr natural `B`, care reprezintă numărul de plecări din orașul `B`. Fiecare dintre următoarele `B` linii conține timpul unei plecări din orașul `B`. Timpii de plecare sunt dați în ordine cronologică în formatul `HH:MM` (ora și minutul). Dacă ora sau minutul nu este un număr de două cifre, va fi precedat de un 0.

Date de ieșire

Fișierul de ieșire `bac.out` conține o singură linie pe care se află numărul minim de bacuri necesare pentru a respecta programul.

Restricții

- $1 \leq K, L \leq 1000$
- $1 \leq A, B \leq 1440$
- timpii sunt cuprinși între `00:00` și `23:59`

Exemple

bac.in	bac.out	bac.in	bac.out	bac.in	bac.out
30 15 1 08:00 1 08:00	2	90 30 2 09:00 10:00 4 08:00 11:00 14:00 20:00	3	15 30 2 08:00 12:00 1 08:45	1

38. *Numere*

Să se genereze toate şirurile formate din n cifre, fiecare şir generat având următoarele proprietăţi:

– conţine numai cifre din mulţimea {1, 2, 3, 4};

– oricare două cifre alăturate sunt fie ambele pare, fie ambele impare.

Numărul natural n ($3 \le n \le 15$) se citeşte de la tastatură. Toate soluţiile vor fi scrise una după alta, cu spaţii între soluţii, fiecare şir fiind scris fără spaţii între cifrele ce-l formează. De exemplu, pentru n=3, se afişează (nu neapărat în această ordine) şirurile: 111 113 131 133 311 313 331 333 222 224 242 244 422 424 442 444.

(variantă Bacalaureat, august 2003)

39. *Adevăr*

Să considerăm un grup format din N persoane, numerotate de la 1 la N. Aceste persoane pot fi împărţite în două categorii: persoane care spun adevărul (categoria A) şi persoane care mint (categoria F). Toate persoanele din grup au fost chemate la un interviu. Aici, fiecare persoană a nominalizat câţiva membri ai grupului, precizând că aceştia aparţin uneia din cele două categorii.

Scrieţi un program care, pe baza informaţiilor obţinute la interviu, să determine care persoane spun adevărul şi care mint.

Date de intrare

Fişierul de intrare adevar.in conţine pe prima linie N, numărul de persoane din grup. Următoarele 2N linii conţin afirmaţiile persoanelor. Mai exact, afirmaţiile persoanei i sunt plasate pe liniile 2i şi 2i+1 sub forma:

K A_1 A_2 ... A_K

L B_1 B_2 ... B_L

Ceea ce înseamnă că persoana i a spus că persoanele A_1 A_2 ... A_K spun adevărul şi că persoanele B_1 B_2 ... B_L mint.

Fiecare persoană face cel puţin o afirmaţie despre o altă persoană şi nici o persoană nu face afirmaţii despre ea însăşi. De asemenea, nici o persoană nu va afirma că o altă persoană spune adevărul, dar şi minte în acelaşi timp.

Dacă o persoană spune adevărul, atunci toate afirmaţiile persoanei respective trebuie să fie adevărate. Dacă o persoană minte, cel puţin una dintre afirmaţiile persoanei respective trebuie să fie falsă.

Date de ieşire

Fişierul de ieşire adevar.out conţine N linii. Pe linia i se precizează dacă persoana i spune adevărul sau minte. Dacă persoana i spune adevărul, pe linia i se va afişa 0, altfel se va afişa 1.

Exemple

adevar.in	adevar.out	adevar.in	adevar.out
3 1 3 0 0 1 1 1 1 0	0 1 0	4 1 2 1 3 1 4 0 0 1 0 1 3	0 0 1 0

40. *Schi*

Un concurs de schi se desfăşoară astfel:

- fiecare cursă are două runde;
- în runda 1, N schiori, numerotaţi de la 1 la N, iau startul în această ordine; când primul schior din prima runda (numerotat cu 1) termină cursa, avem timpul de care schiorul a avut nevoie pentru a parcurge traseul;
- pentru fiecare schior ce urmează dispunem de diferenţa de timp dintre timpul său şi timpul schiorului care este în acel moment lider al cursei, deci cu cel mai bun timp;
- doar cei mai buni M schiori (ca timp) se califică pentru runda 2 şi ei vor porni în cursă în ordinea descrescătoare a timpului obţinut în runda 1 (ultimul schior calificat pentru runda 2 va porni primul, în timp ce primul clasat în runda 1 va porni ultimul);
- în runda 2 vom dispune iniţial de timpul total al primului schior care pleacă în cursă (suma dintre timpul realizat în runda 1 şi timpul realizat în runda 2), iar apoi pentru fiecare dintre următorii schiori vom avea diferenţa de timp dintre timpul său total şi timpul total al schiorului cu cel mai bun timp până în acel moment.

Scrieţi un program care va determina învingătorii după terminarea cursei. Presupunem că nu se poate întâmpla ca doi schiori să obţină acelaşi timp, nici după runda 1, nici după terminarea cursei.

Date de intrare/ieşire

Prima linie a fişierului de intrare `schi.in` conţine două valori întregi N şi M separate printr-un singur spaţiu, reprezentând numărul total de schiori, respectiv numărul de schiori calificaţi pentru runda 2. Linia a doua conţine timpul primului schior din runda 1, apoi următoarele `N-1` linii conţin diferenţele de timp pentru ceilalţi `N-1` schiori din runda 1. Următoarea linie conţine timpul primului schior după runda 2, apoi, pe următoarele `M-1` linii, diferenţele de timp pentru ceilalţi `M-1` schiori după runda 2.

Prima linie a fişierului de ieşire `schi.out` va conţine numărul de ordine al schiorului clasat pe primul loc (aur), linia a doua, numărul de ordine al schiorului clasat pe locul 2 (argint), iar linia a treia, numărul de ordine al schiorului clasat pe locul 3 (bronz).

Restricţii

- $3 \leq M \leq N \leq 100$
- timpul pentru orice cursă este între `10` şi `300` de secunde
- timpii sunt numere reale cu cel mult două cifre zecimale

Exemple

schi.in	schi.out	schi.in	schi.out
3 3 25.13 +1.14 +2.18 45.08 +2.14 +3.11	3 2 1	4 3 29.18 +2.18 +0.05 +1.13 54.22 +1.23 +1.11	4 1 3

41. *Rima*

O poezie constă din `N` strofe ($1 \leq N \leq 5$), fiecare strofă având patru linii. Fiecare linie este constituită din unul sau mai multe cuvinte separate prin spaţii şi/sau semne de punctuaţie (`' : ; . ! ?`). Fiecare cuvânt este format din una sau mai multe litere ale alfabetului englez. Lungimea maximă a unei linii este de 100 de caractere.

Vom spune că ultima silabă a unui cuvânt este secvenţa de caractere care începe cu ultima vocală din cuvânt până la sfârşitul cuvântului. În cazul în care cuvântul nu are vocale, ultima silabă este chiar cuvântul întreg.

Două linii rimează dacă au aceeaşi ultimă silabă, ignorând diferenţa dintre literele mari şi mici. Strofele pot avea rimă perfectă, rimă uniformă, rimă încrucişată, rimă imbricată sau rimă albă.

Strofa are rimă perfectă dacă toate cele patru linii rimează (`a a a a`). Dacă strofa nu are o rimă perfectă, atunci se poate spune că are:

- rimă uniformă, dacă prima şi a doua linie rimează şi, la fel, linia a treia cu a patra (`a a b b`);
- rimă încrucişată, dacă prima şi a treia linie rimează şi, la fel, linia a doua cu a patra (`a b a b`);
- rimă imbricată, dacă prima şi a patra linie rimează şi, la fel, linia a doua cu a treia (`a b b a`);
- rimă albă, dacă nu este nici unul dintre cazurile anterioare.

Scrieți un program care determină pentru fiecare dintre strofele poeziei tipul rimei.

Date de intrare/ieșire

Prima linie a fișierului de intrare `rima.in` conține numărul natural `N` (numărul de strofe din poezie). Următoarele `4N` linii ale fișierului de intrare conțin liniile poeziei. Fișierul de ieșire `rima.out` va conține `N` linii. Pe fiecare linie a fișierului se va scrie unul dintre cuvintele `perfecta`, `uniforma`, `incrucisata`, `imbricata`, `alba`, care vor indica tipul rimei din strofa respectivă.

Exemple

rima.in	rima.out
1 A fost odata ca-n povesti A fost ca niciodata Din rude mari imparatesti O prea frumoasa fata	incrucisata
2 Vreti sa va spun cine sunt eu Un pod cu chip de curcubeu Un curcubeu multicolor Armonizat pe placul tuturor Dar intr-o zi o fata bat-o focul Mi-a-ntors din cale pasul obosit Unde-as fi fost de nu m-as fi oprit Si nu mi-as fi vandut ei tot norocul	uniforma imbricata

5. Stiva şi coada

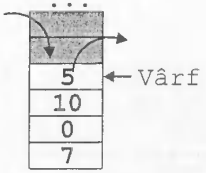
5.1. *Stiva*

Stiva este o structură de date abstractă pentru care atât operaţia de inserare a unui element în structură, cât şi operaţia de extragere a unui element se realizează la un singur capăt, denumit *vârful* stivei.

Singurul element din stivă la care avem acces direct este cel de la vârf.

Operaţii caracteristice

Singurele operaţii care pot fi executate cu o stivă sunt:



- crearea unei stive vide;
- inserarea unui element în stivă (operaţie denumită în literatura de specialitate PUSH[30]);
- extragerea unui element din stivă (operaţie denumită în termeni de specialitate POP[31]);
- accesarea elementului de la vârf (operaţie denumită TOP[32]).

Ca să ne imaginăm mai bine funcţionarea unei stive, să ne gândim cum lucrăm cu un teanc de farfurii. Când dorim să punem o farfurie în teanc, o punem deasupra, când dorim să luăm o farfurie din teanc, o luăm tot pe cea de deasupra. Motivul este lesne de înţeles: nu ne-am propus să spargem farfuriile!

Acest mod de funcţionare face ca ultimul element inserat în stivă să fie primul extras. Din acest motiv, stiva este definită şi ca o structură de date care funcţionează după principiul LIFO (*L*ast *I*n *F*irst *O*ut – ultimul intrat primul ieşit).

---

30. În traducere exactă, PUSH înseamnă „a împinge". Termenul sugerează o imagine plastică: imaginându-ne stiva ca un sac, PUSH ar însemna că împingem înăuntru un element, prin capătul superior (nu prin mijloc sau pe la fundul sacului).
31. În traducere, POP înseamnă „a scoate cu zgomot (de exemplu, dopul etc.), a descărca (pistolul etc.), a ieşi repede sau pe neaşteptate" (Levițchi, Leon; Bantaş, Andrei – Dicţionar englez-român). Imaginea este, de asemenea, sugestivă: POP este operaţia prin care primul element, cel de deasupra, „sare" din structură (sau din sac, ca să păstrăm analogia).
32. În traducere, TOP înseamnă „vârf".

Care este utilitatea stivelor?

În informatică, stiva joacă un rol fundamental. Pentru a înțelege mecanismele fundamentale ale programării (de exemplu, funcțiile sau recursivitatea) este necesară cunoașterea noțiunii de stivă. Pe scurt, stiva este utilă în situații în care este necesară memorarea unor informații și regăsirea acestora într-o anumită ordine, descrisă de principiul LIFO. Stiva este utilizată atunci când programul trebuie să amâne execuția unor operații, pentru a le executa ulterior, în ordinea inversă a apariției lor. Operația curentă este cea corespunzătoare vârfului stivei, în stivă fiind reținute toate informațiile necesare programului pentru a executa operațiile respective.

Cum implementăm o stivă?

Stiva este o structură de date abstractă, ce poate fi implementată în diferite moduri. De exemplu, putem implementa o stivă ca un vector în care reținem elementele stivei. Pentru ca acest vector să funcționeze ca o stivă, singurele operații permise sunt operațiile caracteristice stivei.

Pentru exemplificare, să prezentăm declarațiile necesare pentru implementarea unei stive cu elemente de tip `int`:

```
#define DimMax 100    //numarul maxim de elemente din stiva
typedef int Stiva[DimMax];
//tipul Stiva implementat ca vector
Stiva S;                    //stiva
int   vf;                   //varful stivei
```

Crearea unei stive vide

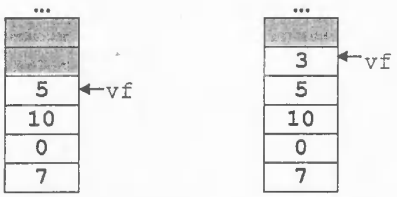
Pentru a crea o stivă vidă inițializăm vârful stivei cu `-1` (vârful stivei indică întotdeauna poziția ultimului element introdus în stivă; elementele sunt memorate în vector începând cu poziția 0):

```
vf=-1;
```

Inserarea unui element în stivă

Pentru a insera un element `x` în stiva `S` trebuie să verificăm în primul rând dacă „avem loc", deci dacă stiva nu este plină. Dacă stiva este plină, inserarea nu se poate face, altfel vom mări vârful stivei și vom plasa la vârf noul element.

De exemplu, dacă dorim să inserăm elementul x = 3 în stiva din figura următoare, obţinem:



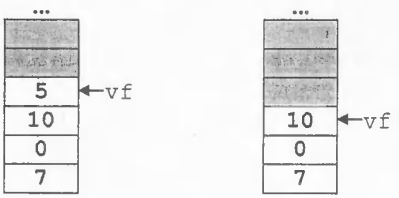
Stiva S înainte de inserare Stiva S după inserare

```
if (vf == DimMax-1)       //stiva este plina
    cout<<"Eroare - stiva este plina\n";
    else                  //inseram elementul x in stiva S
    S[++vf] = x;
```

Extragerea unui element din stivă

Pentru a extrage un element dintr-o stivă S trebuie să verificăm în primul rând dacă există elemente în stivă (deci dacă stiva nu este vidă). Dacă da, reţinem elementul de la vârful stivei într-o variabilă (să o notăm x), după care micşorăm cu o unitate vârful stivei.

De exemplu, dacă extragem un element din stiva din figura următoare, obţinem:



Stiva S înainte de extragere Stiva S după extragere

```
if (vf<0)                      //stiva este vida
   cout<<"Eroare - stiva este vida\n";
   else                       //extragem elementul de la varf
   x = S[vf--];
```

Accesarea elementului de la vârf

Prin modul său restrictiv de funcţionare, stiva permite numai accesarea elementului de la vârf. Dacă dorim să aflăm valoarea unui alt element al stivei, ar trebui să „golim" stiva (deci să extragem succesiv elemente) până la elementul dorit.

Accesarea elementului de la vârf presupune determinarea valorii acestuia, valoare pe care noi o vom reține într-o variabilă denumită `x`.

```
x = S[vf];
```

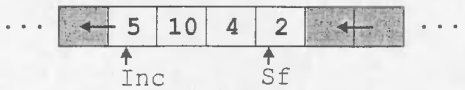
Observații

1. Dezavantajul implementării unei stive ca vector alocat static constă în faptul că indiferent de numărul de elemente existente în stivă, dimensiunea zonei de memorie alocată stivei este aceeași (`DimMax`).
2. Pentru a executa operații cu stiva alocată static este suficient să cunoaștem vârful stivei. Ca să reținem mai ușor modul de funcționare a stivei, ne imaginăm că la inserare vârful stivei urcă, iar la extragere vârful coboară.

5.2. *Coada*

Coada este o structură de date abstractă pentru care operația de inserare a unui element se realizează la un capăt, în timp ce operația de extragere a unui element se realizează la celălalt capăt.

Singurul element din coadă la care avem acces direct este cel de la început.



Operații caracteristice

Singurele operații ce pot fi executate cu o coadă sunt:

- crearea unei cozi vide;
- inserarea unui element în coadă;
- extragerea unui element din coadă;
- accesarea unui element.

Executarea acestor operații asupra unei cozi presupune cunoașterea începutului cozii (să-l notăm `Inc`) și a sfârșitului acesteia (să-l notăm `Sf`).

Modul de funcționare a unei cozi este foarte ușor de intuit: toată lumea a „stat la coadă", măcar o dată. Orice situație în care sunt mai multe cereri de acces la o resursă unică (de exemplu, mai mulți clienți și o singură vânzătoare; o singură pompă de benzină și mai multe mașini, un singur pod și mai multe capre etc.) necesită formarea unei „linii de așteptare". Dacă nu apar alte priorități, cererile sunt satisfăcute în ordinea sosirii.

Datorită faptului că întotdeauna este extras („servit") primul element din coadă, iar inserarea oricărui nou element se face la sfârșit („la coadă"), coada este definită ca o structură de date care funcționează după principiul *FIFO* (*F*irst *I*n *F*irst *O*ut – primul intrat primul ieșit).

Care este utilitatea unei cozi?

Utilitatea structurii de tip coadă reiese din modul său de funcționare – este necesară utilizarea unei cozi atunci când informațiile trebuie prelucrate exact în ordinea în care „au sosit" și ele sunt reținute în coadă până când pot fi prelucrate. În informatică, cozile sunt utilizate frecvent. De exemplu, să considerăm că avem o rețea de calculatoare și o singură imprimantă. Când utilizatorii rețelei vor da comenzi de tipărire, imprimanta nu poate răspunde tuturor comenzilor în același timp (imaginați-vă ce-ar ieși!). Prin urmare, comenzile de tipărire primite sunt înregistrate într-o coadă (*Print Queue* – coadă de tipărire). Imprimanta va tipări documentele pe rând, în ordinea în care au fost înregistrate în coadă. Un alt exemplu: pentru a mări viteza de execuție, microprocesoarele Intel utilizează o coadă de instrucțiuni în care sunt memorate instrucțiunile ce urmează a fi executate.

Cum implementăm o coadă?

Coada este o structură de date abstractă care poate fi implementată în diferite moduri. Ca și în cazul stivei, coada poate fi implementată static, reținând elementele sale într-un vector.

Să considerăm următoarele declarații care reprezintă o coadă cu elemente de tip `int` alocată static:

```
#define DimMax 100    //numarul maxim de elemente din coada
typedef int Coada[DimMax];
//tipul Coada implementat ca vector
Coada C;                      //coada
int  Inc, Sf;                 //inceputul si sfarsitul cozii
```

Elementele cozii sunt memorate în vector de la poziția `Inc` până la poziția `Sf`, deci numărul lor este `Sf-Inc+1`.

Crearea unei cozi vide

Pentru a crea o coadă vidă trebuie să inițializăm variabilele `Inc` și `Sf` astfel:

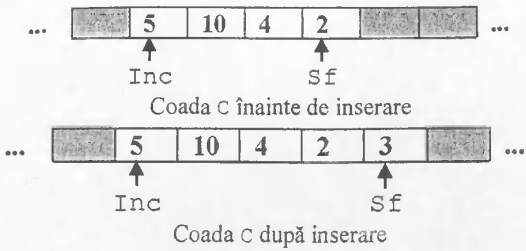
```
Inc = 0;
Sf = -1;
```

Observați că numărul de elemente din coadă este `0`, iar poziția pe care va fi plasat primul element din coadă este `0`.

Inserarea unui element în coadă

Pentru a insera un element `x` în coada `C` trebuie să verificăm în primul rând dacă „avem loc", deci dacă variabila `Sf` nu depășește dimensiunea maximă a vectorului. Dacă inserarea se poate face, vom mări valoarea variabilei `Sf` cu o unitate (coada „crește") și vom plasa la sfârșit noul element.

De exemplu, să inserăm elementul x = 3 în coada din figura următoare:

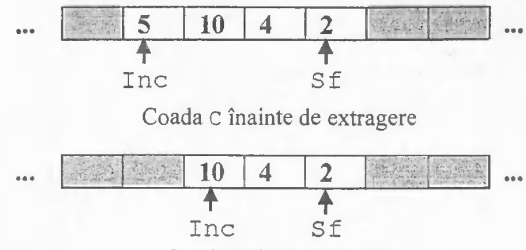



```
if (Sf == DimMax-1)          //nu mai avem loc
   cout<<"Eroare\n";
   else                      //inseram elementul x in coada C
   C[++Sf] = x;
```

Extragerea unui element din coadă

Pentru a extrage un element dintr-o coadă C trebuie să verificăm în primul rând dacă numărul de elemente din coadă este diferit de 0 (coada nu este vidă). Dacă da, reținem elementul de la începutul cozii într-o variabilă (să o notăm x), după care mărim cu o unitate începutul cozii.

De exemplu, să extragem un element din coada din figura următoare:




```
if (Inc > Sf)            //coada este vida
 cout<<"Eroare \n";
 else                    //extragem primul element
 x = C[Inc++];
```

Accesarea unui element

Singurul element al unei cozi care poate fi accesat direct este primul. Dacă dorim să aflăm valoarea unui alt element, va trebui să extragem succesiv elemente până la cel dorit.

Accesarea primului element are ca scop determinarea valorii acestuia, valoare pe care o vom reține într-o variabilă denumită x.

```
x = C[Inc];
```

Observație

În acest mod de alocare statică a unei cozi observați că, pe măsură ce inserăm şi extragem elemente, atât sfârşitul, cât şi începutul cozii „migrează" către limita maximă a vectorului. Dezavantajul este că în vector rămân poziții neutilizate (de la `0` până la `Inc-1`). De exemplu, ar putea să apară o situație în care coada conține un element, dar operația de inserare să nu fie posibilă pentru că `Inc=Sf=DimMax-1`. O soluție mai bună ar fi de a reutiliza circular spațiul de memorie alocat cozii. Pentru aceasta, următoarea poziție după poziția `i` poate fi:

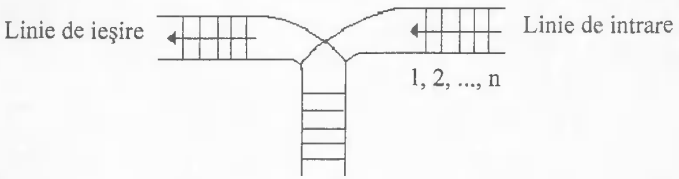
- `i+1`, dacă `i<DimMax -1`;
- `0`, dacă `i=DimMax-1`.

Acest lucru se poate scrie concis: `(i+1)%DimMax`.

5.3. *Aplicații*

Depou

Să considerăm următorul sistem de manevrare a vagoanelor într-un depou:



Observați că în dreapta sunt vagoane, numerotate crescător, de la `1` la `n`. În orice moment, din dreapta poate intra un vagon în depou sau un vagon poate părăsi, prin stânga, depoul. Construiți un tren astfel încât vagoanele să fie numerotate în ordine inversă (de la `n` la `1`).

Soluție

Observați că depoul funcționează ca o stivă: întotdeauna ultimul vagon intrat va fi primul vagon ieşit.

Prin urmare, va fi suficient să introducem în depou toate vagoanele, în ordinea sosirii, apoi să le extragem pe toate.

De exemplu, dacă în depou intră vagonul 1, apoi vagonul 2, apoi vagonul 3, vor ieşi în ordine vagoanele 3, 2, 1.

```
#include <iostream.h>
#define NrMaxVagoane  100
int S[NrMaxVagoane], n, Vf = 0;

int main ()
{ int i;
  cout << "Introduceti numarul de vagoane n = "; cin >> n;
  if (n > NrMaxVagoane)
     cout << "Prea multe vagoane! Nu incap in depou! \n";
     else
     {
       //introducem cele n vagoane in depou
       for (i=1; i<=n; i++) S[++Vf] = i;
       //formam trenul, extragand vagoanele din depou
       while (Vf) cout << S[Vf--] << ' ';
      }
  return 0;}
```

Exerciţii

1. Să presupunem acum că doriţi să construiţi un tren mai special: vagoanele cu număr par vor merge la un moment dat în altă direcţie faţă de cele cu număr impar. Din acest motiv, toate vagoanele cu număr par trebuie să fie plasate la începutul trenului (în ordinea crescătoare a numerelor lor), iar cele cu număr impar, la sfârşitul trenului, în ordinea descrescătoare a numerelor lor. Cum faceţi?

2. Calculaţi pentru n=3 şi pentru n=4 care sunt toate permutările de vagoane ce se pot realiza în depou. Există permutări ce nu se pot realiza în acest mod, mai exact utilizând o stivă?

Observaţie

Problema precedentă ilustrează foarte clar o proprietate esenţială a stivei: stiva inversează ordinea elementelor introduse.

Paranteze

Se citeşte pe o linie de la tastatură o succesiune de paranteze rotunde deschise şi închise, până la întâlnirea caracterului '.'. Întâlnirea unei paranteze deschise determină introducerea acesteia într-o coadă. Întâlnirea unei paranteze închise determină extragerea unui element din coadă. Verificaţi dacă parantezele din şirul citit se închid corect şi determinaţi dimensiunea maximă a cozii (numărul maxim de paranteze deschise reţinute în coadă la un moment dat).

Exemple

Şirul de intrare	Rezultate
(()).	Parantezele se inchid corect. Dimensiunea maxima a cozii este 2.
((().	Parantezele nu se inchid corect. Dimensiunea maxima a cozii este 3.
()).	Parantezele nu se inchid corect. Dimensiunea maxima a cozii este 1.

Soluţie

Vom citi şirul de intrare caracter cu caracter, până când întâlnim caracterul '.'. Pentru fiecare caracter citit verificăm dacă este paranteză închisă sau deschisă.

Dacă este paranteză deschisă, o inserez în coadă, şi fiindcă dimensiunea cozii creşte, verific dacă dimensiunea curentă este mai mare decât dimensiunea maximă de până acum, caz în care reţin dimensiunea curentă ca maximă.

Dacă este paranteză închisă, trebuie să extrag o paranteză deschisă din coadă. Pentru ca extragerea să fie posibilă trebuie ca în coadă să existe paranteze închise (altfel, tragem concluzia că parantezele nu se închid corect şi nu putem face extragerea).

După citirea tuturor caracterelor din şirul de intrare, verificăm dacă în coadă mai sunt paranteze deschise. În acest caz, deducem că parantezele nu se închid corect (au fost mai multe paranteze deschise decât închise).

```
#include <iostream.h>
#define DimMaxCoada 100
int C[DimMaxCoada];
int Inceput, Sfarsit=-1, LgMax, Corect=1;
//Corect este 1 daca parantezele se inchid corect
char p;
int main ()
{
 do                    //citesc caracterele, unul cate unul
   { p=cin.get();
     if (p == '(')
        { //inserez o paranteza deschisa in coada
          C[++Sfarsit] = '(';
          if (Sfarsit-Inceput+1>LgMax)
             LgMax=Sfarsit-Inceput+1;
        }
        else
        if (p == ')')
           //verific daca pot extrage un element din coada
           if (Inceput <= Sfarsit)
              Inceput++;      //extrag un element din coada
            else Corect=0;//parantezele nu se inchid corect
   }
 while (p != '.');
```

```
if (Inceput <= Sfarsit) Corect = 0;
//in coada au mai ramas paranteze deschise
if (Corect) cout << "Parantezele se inchid corect.\n";
    else cout << "Parantezele nu se inchid corect.\n";
cout << "Dimensiunea maxima a cozii este " << LgMax<<endl;
return 0;}
```

Manna-Pnueli

Calculați pentru x întreg, citit de la tastatură, valoarea funcției *Manna-Pnueli*:

$$f : Z \to Z$$

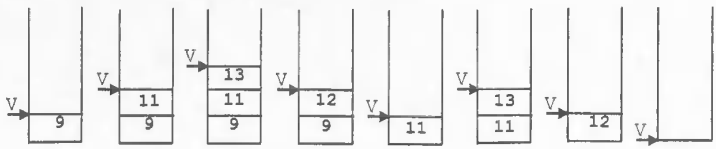
$$f(x) = \begin{cases} x - 1, \text{dacă } x \geq 12 \\ f(f(x+2)), \text{dacă } x < 12 \end{cases}$$

Soluție

Funcția este definită într-un mod special: în cazul în care argumentul x este mai mare sau egal cu 12, valoarea funcției se poate calcula direct. Dacă argumentul x este mai mic decât 12, valoarea funcției nu se poate calcula direct, ci trebuie calculată valoarea f(f(x+2)). Observați că în definiția funcției f intervine însăși funcția f. O astfel de definiție se numește în matematică *definiție recurentă*[33].

Prin urmare, dacă argumentul x este mai mic decât 12, nu putem calcula imediat valoarea funcției f, trebuie să amânăm acest calcul, pe care îl vom putea face numai după ce calculăm f(f(x+2)). Dar pentru a calcula f(f(x+2)) trebuie să calculăm mai întâi f(x+2). Evaluarea funcției presupune amânarea unor operații și executarea lor în ordine inversă. Din acest motiv este necesar să utilizăm o stivă în care să reținem argumentele funcției f, în ordinea în care intervin în procesul de evaluare. Când putem calcula direct valoarea funcției pentru argumentul de la vârful stivei, extragem din stivă valorile argumentelor corespunzătoare ultimelor două apeluri și inserăm valoarea calculată a funcției.

De exemplu, să urmărim pentru x=9 cum sunt memorate în stivă argumentele pentru care trebuie evaluată funcția:



33. La informatică o vom numi ulterior recursivă.

Obţinem f(x)=11.

```
#include <iostream.h>
#define DimMax 100
int  S[DimMax], x, y, V;
int main()
{cout << " x= "; cin >> x;
 S[V] = x; //initializam stiva cu parametrul primului apel
 while (V >= 0)
       {y = S[V];
        if (y >= 12) //calculam valoarea functiei direct
           {//plasam in stiva valoarea calculata a functiei
           if (--V >= 0) S[V] = y-1;           }
           else
           /* functia nu poate fi calculata direct,
           inseram in stiva si parametrul noului apel */
           S[++V] = y+2;
        }
 cout << "f (" << x << ")= " << y-1; return 0;}
```

Caroiaj

Se consideră un caroiaj dreptunghiular cu m linii şi n coloane, în care pe anumite poziţii sunt plasate obstacole. În poziţia iniţială (x_0, y_0) se află plasat un mobil. Să se determine, pentru toate poziţiile în care mobilul poate ajunge, distanţa minimă de la poziţia iniţială a mobilului, măsurată în deplasări elementare. Prin deplasare elementară se înţelege deplasarea mobilului cu o poziţie stânga, dreapta, sus sau jos.

Soluţie

Reprezentăm caroiajul ca o matrice A, în care marcăm obstacolele cu -1, iar poziţiile libere cu -2. La sfârşitul algoritmului, în matricea A vom avea:

A[i][j]= -1, dacă poziţia (i,j) este obstacol;

A[i][j]= -2, dacă poziţia (i,j) nu este accesibilă din (x_0, y_0);

A[i][j]= distanţa minimă de la (x_0, y_0) la (i,j), altfel.

Pentru a nu testa la fiecare pas dacă nu am ajuns la o margine a caroiajului, vom borda caroiajul cu obstacole (linia şi coloana 0, respectiv linia n+1 şi coloana m+1 le vom utiliza ca bordură).

Ideea constă în a calcula distanţa minimă până la poziţiile în care se poate ajunge dintr-o singură mutare, apoi la poziţiile la care se poate ajunge din două mutări etc. Pentru a reţine poziţiile în ordinea distanţei lor faţă de poziţia iniţială, vom folosi o coadă C, în care vom reţine, pentru fiecare poziţie atinsă, coordonatele şi distanţa minimă de la poziţia iniţială.

```
#include <fstream.h>
#include <iomanip.h>
#define DimMax 20
#define DimMaxCoada 400
int dx[4]={-1, 0, 1, 0}, dy[4]={0, 1, 0, -1};
//deplasarile pe linie si coloana pe directiile N,E,S,V
struct Element
  {int l,c;    //pozitia in caroiaj
   unsigned d; //distanta minima minima pana la pozitia l,c
  } C[DimMaxCoada], x, y;
int A[DimMax][DimMax], n, m, x0, y0, i, j, k, IncC, SfC;
int main()
{ifstream fin("careu.in");
 fin >> n >> m;          //citesc dimensiunile caroiajului
 fin >> x0 >> y0;        //citesc pozitia initiala
 //marchez cu -2 pozitiile libere
 for (i=1;i<=n;i++)
     for (j=1;j<=m;j++) A[i][j] = -2;
 while (fin)
       {fin >> i >> j;   //citesc coordonatele obstacolelor
        A[i][j] = -1; }  //marchez obstacolele cu -1
 fin.close();
 //bordez caroiajul cu obstacole
 for (i=1;i<=n;i++) A[i][0]=A[i][m+1]=-1;
 for (i=1;i<=m;i++) A[0][i]=A[n+1][i]=-1;
 //initializez coada
 x.l=x0; x.c=y0; x.d=0; A[x0][y0]=0;      //pozitia initiala
 C[IncC]=x;
 while (IncC <= SfC)                      //parcurg caroiajul
       {
        x = C[IncC++];          //extrag un element din coada
        //ma deplasez in cele patru directii posibile
        for (k=0; k<4; k++)
            { y.l = x.l + dx[k];  y.c = x.c + dy[k];
                    //y - urmatoarea pozitie in directia k
            if (A[y.l][y.c] == -2)
               {
         //y- pozitie libera cu distanta minima necalculata
                  y.d = x.d + 1; A[y.l][y.c] = y.d;
                  //inserez pozitia y in coada
                  C[++SfC]= y;
               }
            }
        }
 //afisez solutia
 ofstream fout("careu.out");
 for (i=1;i<=n;i++)
     { for (j=1;j<=m;j++) fout<<setw(3)<<A[i][j];
       fout<<endl; }
 fout.close();
 return 0;}
```

De exemplu, pentru fişierul de intrare:

```
5 5                    //dimensiunile caroiajului
3 3                    //pozitia initiala
1 2                    //obstacole
1 3
2 4
4 3
4 4
2 1
```

obţinem soluţia:

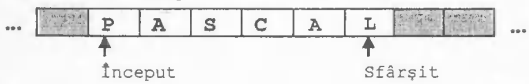
```
-2 -1 -1  5  4
-1  2  1 -1  3
 2  1  0  1  2
 3  2 -1 -1  3
 4  3  4  5  4
```

5.4. *Probleme propuse*

1. Determinaţi valoarea de adevăr a următoarelor propoziţii:
   a. Coada este un vector în care putem adăuga elemente doar la sfârşit şi putem extrage elemente doar de la început.
   b. Extragerea elementelor dintr-o stivă se realizează în ordinea inversă a introducerii lor.
   c. Coada este o structură de date care funcţionează după principiul LIFO.
   d. Afişarea elementelor dintr-o stivă se poate face prin parcurgerea vectorului în care sunt memorate elementele stivei, începând de la vârf.

2. Să considerăm o stivă iniţial vidă, care poate conţine litere. Ilustraţi efectul următoarelor operaţii: inserează `R`, inserează `A`, inserează `M`, inserează `A`, extrage un element, extrage un element, inserează `V` (exprimate în mod echivalent – `PUSH(R); PUSH(A); PUSH(M); PUSH(A); POP; POP; PUSH(V)`).

3. Să considerăm coada din figura următoare:



Ilustraţi efectul următoarelor operaţii: extrage un element, extrage un element, inserează `C`, extrage un element, inserează `U`, inserează `L`.

4. Implementaţi operaţiile elementare de lucru cu o coadă alocată static, utilizând circular spaţiul de memorie.

5. *Simulare*

Multe fenomene fizice, chimice sau biologice nu pot fi observate direct (fie condiţiile nu permit o observaţie directă, fie observarea lor implică probleme practice

sau financiare majore). În astfel de situații se recurge la o simulare – un program care imită evoluția fenomenului respectiv.

Să analizăm un exemplu simplu: simularea activității într-un cabinet stomatologic în care există un singur medic, care lucrează de la 9 la 17. Fiecare pacient care sosește la medic ia loc pe un scaun și așteaptă până când medicul devine disponibil și asistenta îl invită în cabinet. Presupunând că medicul tratează pacienții în ordinea sosirii lor, fără alte priorități (obiective sau subiective), simulați modul de desfășurare a activității! În urma acestei simulări, determinați numărul minim de scaune necesar pentru pacienții care așteaptă, durata medie de așteptare, precum și durata medie a unui tratament.

Pentru simulare, presupunem că am generat aleator datele de intrare în fișierul `'PACIENTI.IN'`. Fișierul conține câte o linie pentru fiecare pacient, pacienții fiind memorați în ordinea sosirii. Pentru fiecare pacient sosit la cabinet va fi reținut timpul la care a sosit (exprimat în număr de minute, calculat de la ora 9, când medicul stomatolog își începe programul) și durata tratamentului (exprimată, de asemenea, în număr de minute).

De exemplu, dacă fișierul de intrare conține:

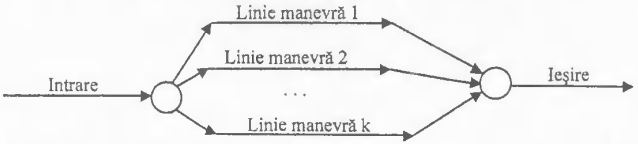
```
10 45
20 30
22 10
```

deducem că la cabinet au sosit trei pacienți:

- primul pacient a sosit la 10 minute de la începerea programului, iar tratamentul lui a durat 45 de minute;
- al doilea pacient a sosit la 20 de minute de la începerea programului, iar tratamentul lui a durat 30 de minute;
- al treilea pacient a sosit la 22 de minute de la începerea programului, tratamentul lui durând 10 minute.

6. *Trenulețe*

Într-un joc cu trenulețul, dintr-o eroare de proiectare, liniile pot fi montate doar într-un singur mod: o linie de intrare din care se desprind într-un nod `k` linii de manevră ce se vor uni în alt nod într-o linie de ieșire, ca în figură:



Jucându-se, un copil așază pe linia de intrare o garnitură de tren cu `n` vagoane numerotate de la `1` la `n`, așezate într-o ordine oarecare. Vagoanele intră câte unul pe liniile de manevră și se deplasează într-un singur sens, de la intrare spre ieșire.

Copilul doreşte să obţină garnitura de tren cu care se joacă în ordinea crescătoare a numerelor vagoanelor. Ajutaţi-l!

Date de intrare

Din fişierul text `tren.in` se citesc de pe prima linie două numere naturale separate printr-un spaţiu `n k` (`1≤n, k≤200`), unde `n` reprezintă numărul de vagoane, iar `k` numărul de linii de manevră. De pe cea de-a doua linie se citesc `n` numerele naturale distincte cuprinse între `1` şi `n`, separate prin spaţii, reprezentând ordinea în care intră cele `n` vagoane pe linia de intrare.

Datele de ieşire

În fişierul text `tren.out` se vor afişa mutările executate sub forma:

X_1 V_1 L_1 P_1
X_2 V_2 L_2 P_2
X_3 V_3 L_3 P_3
...
X_p V_p L_p P_p

unde: X_i poate fi caracterul `M` sau `O`, cu semnificaţia `M` – mutarea se face de pe linia de intrare pe linie de manevră; `O` – mutarea se face de pe linie de manevră pe linia de ieşire; V_i reprezintă numărul vagonului; L_i este linia de manevră pe/de pe care se mută vagonul; P_i este poziţia din linie pe care se mută vagonul. Dacă nu există soluţie, fişierul de ieşire conţine mesajul: `VAGOANELE NU SE POT MUTA`

Exemplu

tren.in	tren.out
5 3	M 1 1 1
1 3 5 4 2	O 1 1 1
	M 3 1 1
	M 5 2 1
	M 4 1 2
	M 2 3 1
	O 2 3 2
	O 3 1 3
	O 4 1 4
	O 5 2 5

7. *Labirint*

Se dă un labirint dreptunghiular de dimensiuni `n` x `m` (`n, m≤100`). Poziţiile din stânga sus şi dreapta jos sunt marcate cu `0`, celelalte conţin unul dintre numerele `1`, `2`, `3`, `4`. Scopul este de a parcurge labirintul din colţul stânga sus până la colţul din dreapta jos pe un drum de lungime minimă, pe direcţii paralele cu laturile sale. Drumul urmat trebuie să plece din `0` în `1`, apoi din `1` în `2`, din `2` în `3`, din `3` în `4`, din `4` în `1` etc. Se poate ajunge în poziţia finală din oricare poziţie vecină ei.

Din fişierul de intrare `INPUT.TXT` se vor citi de pe prima linie numerele întregi `n` şi `m`, care reprezintă dimensiunile labirintului, iar de pe următoarele `n` linii câte `m` numere întregi, separate prin spaţiu, reprezentând labirintul. În fişierul de ieşire `OUTPUT.TXT` se va afişa pe prima linie numărul minim de paşi, iar pe cea de-a doua linie, un şir de caractere care reprezintă succesiunea de mişcări de pe cel mai scurt drum din labirint, folosind codificarea: `D` (jos), `U` (sus), `L` (stânga), respectiv `R` (dreapta).

Exemplu

INPUT.TXT	OUTPUT.TXT
5 4	7
0 1 2 3	RRRDDDD
3 2 1 4	
4 1 2 1	
1 4 3 2	
2 3 4 0	

(Balcaniada de Informatică, Cipru, 1996)

8. *Romeo şi Julieta*

În ultima ecranizare a celebrei piese shakespeariene, Romeo şi Julieta trăiesc într-un oraş modern, comunică prin e-mail şi chiar învaţă să programeze. Într-o secvenţă tulburătoare sunt prezentate frământările interioare ale celor doi eroi încercând fără succes să scrie un program care să determine un punct optim de întâlnire. Ei au analizat harta oraşului şi au reprezentat-o sub forma unei matrice cu `n` linii şi `m` coloane, în matrice fiind marcate cu spaţiu zonele prin care se poate trece (străzi lipsite de pericole) şi cu `X` zonele prin care nu se poate trece. De asemenea, în matrice au marcat cu `R` locul în care se află locuinţa lui Romeo, iar cu `J` locul în care se află locuinţa Julietei. Ei se pot deplasa numai prin zonele care sunt marcate cu spaţiu, din poziţia curentă în oricare dintre cele opt poziţii învecinate (pe orizontală, verticală sau diagonale). Cum lui Romeo nu îi place să aştepte şi nici să se lase aşteptat n-ar fi tocmai bine, ei au hotărât că trebuie să aleagă un punct de întâlnire în care atât Romeo, cât şi Julieta să poată ajunge în acelaşi timp, plecând de acasă. Fiindcă la întâlniri amândoi vin într-un suflet, ei estimează timpul necesar pentru a ajunge la întâlnire prin numărul de elemente din matrice care constituie drumul cel mai scurt de acasă până la punctul de întâlnire. Şi cum probabil există mai multe puncte de întâlnire posibile, ei vor să îl aleagă pe cel în care timpul necesar pentru a ajunge la punctul de întâlnire este minim.

Scrieţi un program care să determine o poziţie pe hartă la care Romeo şi Julieta pot să ajungă în acelaşi timp. Dacă există mai multe soluţii, programul trebuie să determine o soluţie pentru care timpul este minim.

Date de intrare

Fişierul de intrare `rj.in` conţine:

- pe prima linie, numerele naturale `n m`, care reprezintă numărul de linii şi, respectiv, de coloane ale matricei, separate prin spaţiu (`1<n,m≤100`); liniile şi coloanele sunt numerotate începând cu `1`;
- pe fiecare dintre următoarele `n` linii se află `m` caractere (care pot fi doar `R`, `J`, `X` sau spaţiu) reprezentând matricea.

Date de ieşire

Fişierul de ieşire `rj.out` va conţine o singură linie pe care sunt scrise trei numere naturale separate prin câte un spaţiu `tmin x y`, având semnificaţia:

- `x y` reprezintă punctul de întâlnire (`x` – numărul liniei, `y` – numărul coloanei);
- `tmin` este timpul minim în care Romeo şi Julieta ajung la punctul de întâlnire.

Exemple

`rj.in`	`rj.out`	**Explicaţie**

`5 8`
`XXR XXX`
`X X X`
`J X X X`
`XX`
`XXX XXXX`
	`4 4 4`	Traseul lui Romeo poate fi: `(1,3)`, `(2,4)`, `(3,4)`, `(4,4)`.
		Deci timpul necesar lui Romeo pentru a ajunge de acasă la punctul de întâlnire este `4`.
		Traseul Julietei poate fi: `(3,1)`, `(4,2)`, `(4,3)`, `(4,4)`.
		Timpul necesar Julietei pentru a ajunge de acasă la punctul de întâlnire este, de asemenea, `4`. În plus, `4` este punctul cel mai apropiat de ei cu această proprietate.

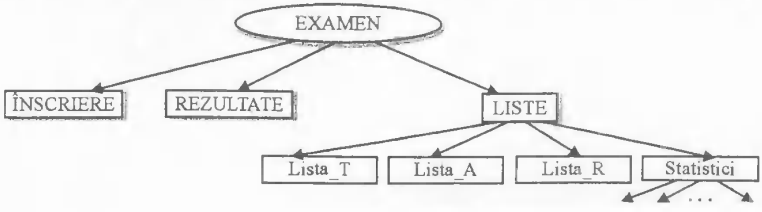
6. Funcții

Pentru proiectarea unor aplicații complexe este necesară descompunerea problemei care trebuie rezolvată în subprobleme relativ independente, pentru fiecare dintre aceste subprobleme scriindu-se module de program mai simple. Cum la orice firmă se lucrează în echipă, modulele de program sunt, de obicei, implementate de mai mulți programatori. Pentru ca programul să funcționeze, în final, modulele de program trebuie asamblate. Prin urmare, într-o etapă prealabilă implementării se face o analiză a problemei de rezolvat, se stabilesc subproblemele și modulele de program care trebuie să rezolve aceste subprobleme, precum și modalitățile în care acestea trebuie să comunice între ele (interfața).

De asemenea, în rezolvarea problemelor apar frecvent operații care „se repetă". În aceste situații ar fi bine să scriem secvența de program corespunzătoare unei astfel de operații o singură dată și să o folosim ori de câte ori este nevoie de ea. În practică, în dezvoltarea unei aplicații nu se pornește de fiecare dată de la zero, ci se utilizează modulele de program deja implementate. Cum ar fi dacă ar trebui să scriem de fiecare dată instrucțiunile care să calculeze radicalul sau care să realizeze comunicarea cu dispozitivele periferice?

Realizarea unui program de complexitate mare impune ca o necesitate organizarea datelor și a prelucrărilor la care acestea trebuie supuse în subprobleme și, corespunzător acestora, sub formă de subprograme.

Să analizăm problema prelucrării datelor în cazul unui examen de admitere. Primul nivel de subprobleme ar putea fi: înscrierea candidaților, înregistrarea rezultatelor probelor, listarea diferitelor situații cerute. Dacă prima și a doua subproblemă sunt relativ simple, cea de-a treia însă, trebuie, la rândul ei, să fie împărțită în subprobleme: obținerea listei generale ordonate alfabetic, a listei celor admiși în ordinea descrescătoare a mediilor, a listei celor respinși ordonate de asemenea alfabetic. În plus, la orice examen sunt solicitate multe alte situații statistice. Deci, schematic:



Dacă luăm în considerație aplicația descrisă mai sus, care este o aplicație reală, utilizată la fiecare examen de admitere în liceu sau într-o instituție de învățământ superior, se observă faptul că întreaga aplicație poate fi împărțită în module și fiecare modul poate fi programat separat.

6.1. *Subprograme în limbajul C/C++*

În limbajele C și C++, subprogramele sunt denumite *funcții*.

Funcțiile reprezintă un element fundamental al limbajului C/C++. Așa cum știm, orice program C/C++ este constituit dintr-o succesiune de funcții, dintre care una este funcția principală, denumită `main()`. La lansarea în execuție a unui program C/C++ este apelată funcția `main()`.

Până acum, în cadrul funcției `main()` am apelat numai funcții standard ale limbajului (de exemplu, `sqrt()`, `clrscr()` etc.). În acest capitol vom învăța să descriem propriile noastre funcții. Acestea vor fi apelate din funcția `main()` sau dintr-o altă funcție apelată din `main()`.

În concluzie, într-un program C/C++ toate prelucrările sunt organizate ca o ierarhie de apeluri de funcții, baza acestei ierarhii fiind funcția `main()`.

Pentru a putea dezvolta și utiliza funcții proprii este necesar să cunoaștem cum se *definesc*, cum se *declară* și cum se *apelează* funcțiile.

6.2. *Definiția unei funcții*

Definiția unei funcții este constituită dintr-un *antet*, care conține numele funcției, tipul rezultatului returnat de funcție și lista parametrilor funcției și *un bloc de instrucțiuni*, care descrie prelucrările efectuate de funcție.

Formatul general al definiției unei funcții este:

```
tip nume(lista_parametri_formali)
{
    declaratii_variabile_locale
    instructiuni
}
```

unde:

`tip` – reprezintă tipul rezultatului returnat de funcție;

`nume` – reprezintă numele funcției;

`lista_parametri_formali` – este constituită din una sau mai multe declarații de parametri formali, separate prin virgulă, sau poate lipsi.

O declarație de parametru formal specifică tipul și numele parametrului, sub forma: `tip nume`.

Parametrii unei funcții constituie o interfață prin intermediul căreia funcția comunică cu exteriorul. Parametrii desemnează date primite de funcție din exterior, date de care depind prelucrările efectuate de funcție.

Parametrii formali sunt denumiți astfel deoarece cu ajutorul lor se descriu în mod *formal* operațiile la care vor fi supuse datele ce vor fi transmise de programul *apelant* spre subprogram.

Observații

Lista de parametri formali poate să fie vidă, dar și în acest caz parantezele trebuie să apară. În standardul ANSI al limbajului C, în cazul în care lista parametrilor formali este vidă, trebuie să se specifice explicit aceasta prin cuvântul rezervat `void`. Deși C++ acceptă o listă vidă de parametri, pentru a păstra compatibilitatea dintre C și C++ este recomandat să utilizăm `void`.

Dacă tipul rezultatului returnat de funcție nu este specificat, implicit este considerat `int`. Dacă funcția nu trebuie să întoarcă nici o valoare, se specifică drept tip al rezultatului `void`.

Exemplul 1

Definim o funcție fără parametri, denumită `scrie`. Funcția afișează un mesaj și nu returnează nici un rezultat:

```
void scrie(void)
{  cout << "Aceasta este prima mea functie C/C++ \n"; }
```

Exemplul 2

Definim o funcție denumită `scrie_suma` care primește ca parametri două numere întregi și afișează suma acestora. Funcția nu returnează nici un rezultat:

```
void scrie_suma(int a, int b)
{  cout << "a+b="<<a+b; }
```

Exemplul 3

Definim o funcție denumită `suma` care primește ca parametri un vector cu maximum `100` de componente întregi și numărul efectiv de elemente din vector. Funcția returnează suma elementelor vectorului:

```
int suma(int a[100], int n)
{ int i, s=0;
  for (i=0; i<n; s+=a[i++]);
  return s; }
```

Observați că funcția utilizează două variabile, `i` și `s`. Deoarece acestea sunt declarate în blocul funcției, ele sunt *variabile locale*.

În blocul funcției intervine instrucțiunea `return`. Această instrucțiune are formatul general:

```
return expresie;
```

Efect

Se evaluează expresia și se încheie execuția funcției, returnând în exterior (mai exact, în funcția apelantă) valoarea expresiei. În cazul în care expresia este vidă, funcția nu returnează nici o valoare.

În concluzie, revenirea dintr-o funcție se face fie după executarea ultimei instrucțiuni (instrucțiunea care precedă acolada închisă a blocului funcției) – caz în care funcția nu returnează nici o valoare, fie la întâlnirea instrucțiunii `return`.

Observație

Pot exista mai multe instrucțiuni `return` în cadrul blocului unei funcții, de exemplu:

```
int max (int a, int b)
{
  if (a>b) return a;
  return b;
}
```

Funcția `max` are doi parametri întregi, `a` și `b`. Dacă `a>b`, atunci funcția returnează valoarea parametrului `a` și execuția funcției se încheie. Dacă nu s-a încheiat execuția funcției (ceea ce echivalează cu `a<=b`), se execută următoarea instrucțiune de după `if`, prin urmare funcția returnează valoarea parametrului `b`. Deci funcția calculează maximul dintre `a` și `b`.

6.3. *Declararea funcțiilor*

Așa cum am văzut în secțiunea precedentă, definiția unei funcții informează compilatorul despre formatul funcției (nume, parametri, tip rezultat) și descrie prelucrările efectuate de funcție.

Declarația unei funcții informează compilatorul despre *existența* funcției și *formatul* acesteia. Mai exact, o declarație de funcție specifică numele funcției, tipul rezultatului returnat de funcție și parametrii acesteia:

```
tip nume(lista_parametri);
```

Observați că o declarație de funcție este constituită din antetul funcției, urmat de `';'`, nu de blocul de instrucțiuni al funcției. Spre deosebire de definiție, în declarație lista parametrilor nu conține în mod obligatoriu și numele parametrilor, ci este permisă doar specificarea tipurilor parametrilor.

Exemple

```
void f1(int, int);
```

Funcția `f1()` are doi parametri de tip `int` și nu întoarce nici un rezultat.

```
long f2(int x, int y[20]);
```

Funcția `f2()` are doi parametri: unul de tip `int` și un tablou de 20 de valori de tip `int` și întoarce ca rezultat o valoare de tip `long int`. În această declarație am specificat și numele parametrilor, nu doar tipul acestora.

```
void clrscr();
```

Funcția `clrscr()` este o funcție declarată în fișierul antet `conio.h` și are rolul de a șterge fereastra text curentă și de a plasa cursorul în colțul din stânga sus al ferestrei. Este o funcție care nu are nici un parametru și nu întoarce nici un rezultat.

```
double sqrt(double);
```

Funcția `sqrt()` are un parametru de tip `double` și returnează radicalul valorii primite ca parametru. Este declarată în fișierul antet `math.h`.

```
int rand (void);
```

Funcția `rand()` este declarată în fișierul antet `stdlib.h` și returnează un număr aleator mai mic decât o constantă `RAND_MAX`, definită de asemenea în `stdlib.h`.

Declarația unei funcții se mai numește și *prototip*.

Utilitatea declarațiilor

Definiția unei funcții apare înaintea apelului funcției numai în cazuri particulare. Acest lucru nu este întotdeauna posibil (de exemplu, dacă funcția nu este definită în același fișier sursă sau este o funcție standard, care se găsește în bibliotecile limbajului în formă compilată).

Totuși, când întâlnește un apel de funcție, compilatorul trebuie să verifice validitatea acestuia. Prin urmare, este necesar ca înaintea oricărui apel de funcție să apară fie definiția funcției, fie declarația acesteia.

Un program C/C++ poate conține pentru o funcție o singură definiție, dar oricâte declarații.

Biblioteci de funcții și fișiere antet

În activitatea de programare există anumite operații care se execută frecvent (de exemplu, citiri, scrieri, sortări, operații matematice complexe, operații cu șiruri de caractere etc.). Pentru aceste operații firmele au dezvoltat funcții specifice. Aceste funcții sunt grupate pe categorii, memorate în formă compilată în colecții, denumite biblioteci. Utilizarea bibliotecilor eliberează programatorii de sarcina de a

scrie și testa un volum mare de cod, întrucât funcțiile din biblioteci[34] sunt deja testate și funcționează corect.

Pentru a utiliza funcțiile dintr-o bibliotecă într-un program, acestea trebuie declarate. Din acest motiv, programatorii creează pentru bibliotecile lor fișiere speciale, denumite fișiere antet sau fișiere *header*, care conțin declarații de funcții și, eventual, declarații de variabile externe. Ca urmare, pentru a utiliza funcțiile dintr-o bibliotecă, este suficient să includem printr-o directivă `#include` fișierul *header* corespunzător. Dacă biblioteca este una standard, plasată în subdirectorul `Include` al directorului în care este instalat mediul de programare, directiva de includere a fișierului antet corespunzător are forma:

```
#include <fisier.h>
```

Dacă biblioteca nu este una standard, ci este creată de voi, directiva de includere a fișierului antet corespunzător are forma:

```
#include "fisier.h"
```

6.4. *Apelul funcțiilor*

Apelul unei funcții se poate realiza în două moduri – printr-o *instrucțiune de apel* sau ca *operand într-o expresie*.

Instrucțiunea de apel al unei funcții are următorul format general:

```
nume(lista_parametri_actuali);
```

unde `nume` reprezintă numele funcției. Lista parametrilor actuali este formată dintr-o succesiune de expresii, separate prin virgulă.

Utilizăm instrucțiuni de apel atunci când funcția apelată nu returnează nici o valoare sau atunci când nu dorim să utilizăm valoarea returnată de funcție, ci ne interesează doar efectuarea prelucrărilor descrise de funcție.

În cazul în care dorim să utilizăm valoarea returnată de funcție ca operand într-o expresie, vom apela funcția în cadrul expresiei astfel:

```
nume(lista_parametri_actuali)
```

Observați că în acest caz lipsește caracterul `';'` care marchează sfârșitul instrucțiunii de apel.

La apelul unei funcții, *valorile* parametrilor actuali sunt atribuite, în ordine, parametrilor formali corespunzători.

---

34. Mediul de programare Borland C++ conține un utilitar special pentru crearea de biblioteci, denumit TLIB. Pentru a afla informații despre modul de utilizare a acestui utilitar este suficient să dați comanda: TLIB ?.

Regulă fundamentală: Parametrii actuali trebuie să corespundă cu parametrii formali ca număr, ordine şi tip

Exemplu

Să considerăm următoarele declaraţii de variabile:

```
int x=4, y=5, m;
```

Putem atribui variabilei `m` maximul dintre `x` şi `y` apelând funcţia `max` definită anterior astfel:

```
m=max(x, y);
```

În acest caz, valoarea parametrului actual `x` înlocuieşte parametrul formal `a` (deci `a` va avea valoarea 4), iar valoarea parametrului actual `y` înlocuieşte parametrul formal `b` (deci `b` va avea valoarea 5).

Pentru a ilustra transferul parametrilor la apel să facem o analogie între un subprogram şi o piesă de teatru. Autorul scrie acţiunea piesei în funcţie de nişte personaje. Personajele care apar în piesă sunt menţionate la începutul piesei, însoţite de o scurtă descriere a „tipului" lor (de exemplu, Conu Leonida – pensionar, 60 de ani; Coana Efimiţa – consoarta lui, 56 de ani; Safta – slujnica lor). În mod similar, autorul unui subprogram descrie acţiunea acestuia în funcţie de nişte parametri formali. În antetul subprogramului sunt precizaţi numele şi tipul parametrilor formali.

Scriind o piesă, un dramaturg descrie formal acţiunea din piesa de teatru. Abia în momentul în care piesa este pusă în scenă ea „prinde viaţă". Fiecare personaj din piesă este interpretat de un actor. Prin urmare, în momentul reprezentaţiei, distribuţia înlocuieşte lista de personaje din piesă. Aceeaşi piesă poate fi jucată cu mai multe distribuţii, însă acţiunea se desfăşoară la fel. În mod analog, parametrii formali ai unui subprogram sunt înlocuiţi la apel („reprezentaţia") de parametrii actuali („distribuţia"). Evident, trebuie să existe o corespondenţă între parametrii formali şi cei actuali, aşa cum trebuie să existe o corespondenţă între personaj şi actor (ar fi destul de dificil să-l distribuim pe Van Damme în rolul Coanei Efimiţa).

Prin urmare, piesa de teatru, cu descrierea personajelor şi a acţiunilor, poate fi asemuită cu definirea unui subprogram cu lista corespunzătoare de parametri formali. Reprezentaţia piesei, când personajele sunt înlocuite de actori, este similară apelului, când parametrii formali sunt înlocuiţi de cei actuali şi acţiunile devin „reale".

Apare întrebarea: *unde sunt memorate valorile parametrilor actuali*?

Funcţiile utilizează o zonă de memorie specială, denumită *stivă* (în limba engleză – *stack*). Denumirea se datorează modului de funcţionare a acestei zone de memorie – LIFO (ultimele informaţii memorate sunt primele informaţii extrase).

La apelarea unei funcţii, se alocă spaţiu de memorie pe stivă pentru valorile parametrilor actuali (*transfer prin valoare*). La încheierea execuţiei funcţiei, zona de memorie alocată pe stivă pentru apel este eliberată

Ca urmare a acestui mod de funcţionare, chiar dacă în blocul funcţiei sunt modificate valorile parametrilor, la ieşirea din funcţie, deoarece memoria alocată pentru parametri se eliberează, valorile modificate *se pierd.*

Exemplu

Să considerăm următoarea funcţie care intenţionează să interschimbe valorile parametrilor x şi y:

```
void schimba(int x, int y)
{
  int aux;
  aux=x; x=y; y=aux;
  cout <<"x="<<x<<" y="<<y<<endl;
}
```

Să presupunem că am declarat variabilele:

```
int a=3, b=4;
```

Secvenţa de instrucţiuni:

```
schimba(a, b);
cout <<"a="<<a<<" b="<<b<<endl;
```

afişează pe ecran:

```
x=4 y=3
a=3 b=4
```

Prin urmare, deşi valorile parametrilor x şi y au fost schimbate în blocul funcţiei, la ieşirea din funcţie valorile variabilelor a şi b care au fost utilizate ca parametri actuali rămân neschimbate. Acest comportament este absolut normal: valorile parametrilor funcţiei au fost copiate pe stivă, copiile au fost interschimbate, apoi la ieşirea din funcţie zona de memorie alocată pe stivă este eliberată, iar copiile s-au pierdut. Funcţia nu poate modifica valorile parametrilor actuali, deoarece nu cunoaşte adresa la care ei sunt memoraţi.

Dacă dorim ca la ieşirea din funcţie să se păstreze valorile modificate ale parametrilor, nu vom transmite ca parametri expresii ale căror valori să fie copiate pe stivă, ci vom transmite *adresele* variabilelor ale căror valori dorim să le modificăm.

Să ilustrăm, printr-o nouă analogie, modul de funcţionare a celor două modalităţi de a transmite parametrii: să presupunem că în oraş trăieşte un Jack Spintecătorul. Cineva doreşte să-mi facă rău, comunicându-i lui Jack o informaţie despre mine.

Informaţia transmisă poate fi, de exemplu, adresa mea. În acest caz, sunt pierdut. Jack îmi cunoaşte adresa, deci poate ajunge la mine. În mod analog, dacă un subprogram cunoaşte adresa unui parametru, modificarea valorii acestui parametru în subprogram se va face la adresa cunoscută şi va avea ca efect modificarea valorii parametrului actual.

Dacă însă informația care ajunge la Jack este fotografia mea, adică o copie, sunt salvat. Orice ar face Jack fotografiei, originalul rămâne nevătămat. Același lucru se întâmplă și în cazul unui parametru transmis prin valoare: în stivă se va face o copie a valorii parametrului, asupra acestei copii se pot face modificări, dar originalul rămâne neschimbat. Mai mult, datorită faptului că stiva este eliberată la revenirea în programul/subprogramul apelant, copia se pierde, deci și modificările făcute asupra ei.

6.5. *Transferul parametrilor prin referință*

În paragraful precedent am văzut că în limbajul C/C++ transferul parametrilor se face prin valoare. Totuși apare frecvent necesitatea de a modifica valorile parametrilor funcției și de a utiliza în afara funcției valorile modificate. În acest caz, este necesar să transmitem funcției adrese (*transfer prin referință*).

Există două modalități de a transmite parametrii prin referință: utilizând *pointeri* (soluție utilizabilă atât în limbajul C, cât și în C++) sau utilizând *referințe* (tipul referință fiind disponibil numai în limbajul C++).

Utilizarea pointerilor ca parametri

Să reluăm funcția `schimba()`, care intenționa să interschimbe valorile parametrilor `x` și `y`, transmițând de data aceasta adresele parametrilor care trebuie modificați:

```
void schimba(int *x, int *y)
{
  int aux;
  aux=*x; *x=*y; *y=aux;
  cout <<"x="<<*x<<" y="<<*y<<endl;
}
```

De data aceasta, dacă presupunem că am declarat variabilele:

```
int a=3, b=4;
```

Secvența de instrucțiuni:

```
schimba(&a, &b);
cout <<"a="<<a<<" b="<<b<<endl;
```

afișează pe ecran:

```
x=4 y=3
a=4 b=3
```

Deci interschimbarea s-a executat cu succes!

Observați că de data aceasta parametrii funcției nu sunt numere întregi, ci pointeri la întregi. La apelul funcției am ținut cont de tipul parametrilor și am transmis adresele variabilelor ale căror valori trebuie să fie interschimbate.

Tipul referință

Tipul referință este o facilitate suplimentară a limbajului C++, introdusă pentru a acoperi o lacună importantă a limbajului C, și anume transmiterea parametrilor prin referință.

O variabilă de tip referință conține adresa unei alte variabile (asemănător unui pointer), dar nu este interpretată ca pointer, ci ca un alt nume asociat variabilei (un alias, un sinonim).

Declararea unei referințe:

```
tip & variabila_referinta=variabila_referita;
```

Declarația de mai sus creează un alt nume (`variabila_referinta`) pentru variabila `variabila_referita` având tipul `tip`.

Se pot crea referințe către variabile de orice tip. Ulterior în program puteți utiliza atât variabila, cât și referința către ea (sinonimul).

Exemplu

```
int x=10;
int &rx=x;
```

Am declarat variabila întreagă `x` și o referință pentru `x`, denumită `rx`. De exemplu, dacă dorim să afișăm valoarea variabilei `x`, putem scrie fie:

```
cout<<x;
```

fie:

```
cout<<rx;
```

Efectul va fi același!

Observații

Valoarea atribuită unei referințe *nu* poate fi schimbată, ea rămâne valabilă pe toată durata execuției blocului de program în care este declarată.

Deși referința seamănă cu o variabilă, ea nu este însăși variabila, ci doar o referire la variabila respectivă (o adresă).

Aparent, referințele nu sunt grozav de utile. Dar să analizăm situația în care utilizăm referințe drept parametri ai unei funcții. Să reluăm funcția `schimba()`, care intenționa să interschimbe valorile parametrilor `x` și `y`, transmițând de data aceasta referințe către parametrii care trebuie modificați:

```
void schimba(int &x, int &y)
{
  int aux;
  aux=x; x=y; y=aux;
  cout <<"x="<<x<<" y="<<y<<endl;
}
```

De data aceasta, dacă presupunem că am declarat variabilele a şi b şi două referinţe către acestea, ra, respectiv rb:

```
int a=3, b=4;
int & ra=a, & rb=b;
```

secvenţa de instrucţiuni:

```
schimba(ra, rb);
cout <<"a="<<a<<" b="<<b<<endl;
```

afişează pe ecran:

```
x=4 y=3
a=4 b=3
```

Deci şi în acest caz interschimbarea s-a executat cu succes!

În concluzie, în cazul utilizării referinţelor drept parametri, beneficiem de toate avantajele pointerilor (posibilitatea de a modifica valorile parametrilor), fără complicaţii de sintaxă.

Evident, orice operaţie cu referinţe se poate realiza şi cu ajutorul pointerilor, afirmaţia reciprocă nefiind valabilă.

Observaţie

Am fi putut apela funcţia schimba() şi sub forma:

```
schimba(a, b);
```

În acest caz, compilatorul va transmite automat adresele variabilelor a şi b, fără a utiliza o variabilă referinţă intermediară. Aceste adrese sunt atribuite la apel parametrilor de tip referinţă corespunzători.

S-ar putea spune că referinţele sunt *pointeri care se dereferenţiază automat.* Pentru compilatorul C++, referinţa este o adresă. Pentru programator, referinţa este doar un alt nume pentru variabilă, deci poate fi interpretată ca valoare a variabilei.

Utilizarea referinţelor este utilă nu numai atunci când un parametru trebuie să îşi modifice valoarea. Transmiterea unui parametru prin valoare presupune copierea valorii parametrului pe stivă (ceea ce uneori poate fi costisitor atât ca spaţiu, cât şi ca timp). În cazul în care variabila transmisă ca parametru este o structură de date voluminoasă, este preferabil să utilizăm o referinţă către variabilă, deoarece pe stivă se va copia doar adresa. Dacă dorim să evităm modificarea valorii unui parametru transmis prin referinţă, putem preceda declaraţia parametrului de specificatorul const, care împiedică modificarea valorii. Orice tentativă de a modifica valoarea unui parametru transmis prin referinţă a cărui declaraţie este precedată de const va fi sancţionată cu eroare la compilare.

De exemplu, dacă modificăm antetul funcţiei schimba() astfel:

```
void schimba (const int & x, int &y);
```

nu putem modifica valoarea parametrului x (atribuirea x=y va genera eroare).

Exemplu

Să urmărim, pas cu pas, pe un exemplu simplu, ce se întâmplă în fiecare moment al execuției unui program care conține un apel de funcție. Pentru identificarea acțiunilor, numerotăm liniile programului.

```
#include <iostream.h>                                          //1
int x, y = 1;                                                  //2
void F(int & a, int b)                                         //3
{                                                              //4
  cout<<"a="<<a<<"  b="<<b<<"  x="<<x<<"  y="<<y<<endl; //5
  a = 5;                                                       //6
  b = 10;                                                      //7
  cout<<"a="<<a<<"  b="<<b<<"  x="<<x<<"  y="<<y<<endl; //8
}                                                              //9
void main(void)                                                //10
{                                                              //11
  F(x, y);                                                     //12
  cout <<"x="<<x<< "  y="<<y<<endl;                            //13
}                                                              //14
```

Urmărind fiecare instrucțiune executabilă, indicăm în tabelul următor linia care se execută, valorile variabilelor globale x și y, valorile depuse pe stivă corespunzătoare parametrilor formali a și b și ce anume se va afișa pe ecran. Semnul '•' semnifică o adresă, săgeata indicând variabila la care se referă adresa.

Linia	Memorie		Stivă			Ecran
	x	y	a	b	adresa revenire	
2	0	1	–	–	–	
12	0	1	•	1	adresa instrucțiunii de pe linia 13	
5	0	1	•	1	adresa instrucțiunii de pe linia 13	a=0 b=1 x=0 y=1
6	5	1	•	1	adresa instrucțiunii de pe linia 13	a=0 b=1 x=0 y=1
7	5	1	•	10	adresa instrucțiunii de pe linia 13	a=0 b=1 x=0 y=1
8	5	1	•	10	adresa instrucțiunii de pe linia 13	a=0 b=1 x=0 y=1 a=5 b=10 x=5 y=1
13	5	1	–	–	–	a=0 b=1 x=0 y=1 a=5 b=10 x=5 y=1 x=5 y=1

Observație

1. Nu uitați că numele unui tablou este un pointer către primul element al tabloului! Prin urmare, când utilizați vectori ca parametri, la definirea/declararea funcției puteți utiliza pentru declararea parametrului una din construcțiile echivalente:

`tip *` , `tip[]` sau `tip[Max]` (unde `tip` reprezintă tipul componentelor vectorului, iar `Max` numărul maxim de elemente din vector). Nu este obligatorie specificarea numărului de elemente din vector între parantezele pătrate.

2. Când apelați o funcție care are parametri de tip tablou, trebuie să transmiteți doar numele tabloului (acesta este el însuși o adresă!).

Exemplu

Scriem un program care citește un vector `a` cu `n` elemente întregi și un vector `b` cu `m` elemente întregi, apoi afișează pe ecran cei doi vectori citiți.

Varianta C:

```
#include <stdio.h>

void  citire_vector(int *, int *);
void  afisare_vector(int, int *);

int main(void)
{ int n, a[10], m, b[10];
  citire_vector(&n, a);
  citire_vector(&m, b);
  afisare_vector(n, a);
  afisare_vector(m, b);
  return 0;             }

void  citire_vector(int * nr, int x[10])
{int i;
 printf("Nr. elemente = "); scanf("%d", nr);
 printf("Elementele vectorului: ");
 for (i=0; i < *nr; i++) scanf("%d", &x[i]);
}
void  afisare_vector(int nr, int a[10])
{int i;
 printf("Elementele vectorului sunt: ");
 for (i = 0; i < nr; i++) printf("%d ", a[i]);
 printf("\n");
}
```

În exemplul precedent am declarat două funcții: `citire_vector()` și `afisare_vector()`. Funcția `citire_vector()` are doi parametri care sunt pointeri cu tipul de bază `int` (`int *`). Observați că la definirea funcției `citire_vector()` am declarat al doilea parametru (vectorul) în mod echivalent, specificând și numărul maxim de componente din vector (`int x[10]`). Primul parametru al funcției `citire_vector()` este numărul de elemente din vector. Este declarat ca pointer, deoarece funcția `citire_vector()` trebuie să transmită în exterior valoarea citită.

La apelul funcțiilor, observați că am precizat doar numele vectorului care se citește, respectiv se afișează. Apelul este corect, numele vectorului fiind un pointer către primul element din vector.

Varianta C++ va utiliza referințe în loc pe pointeri:

```
#include <iostream.h>

void  citire_vector(int &, int *);
void  afisare_vector(int, int *);

int main()
{ int n, a[10], m, b[10];
  citire_vector (n, a);
  citire_vector (m, b);
  afisare_vector(n, a);
  afisare_vector(m, b);
  return 0;            }

void  citire_vector(int & nr, int x[10])
{
 cout << "Nr. elemente = "; cin >> nr;
 cout << "Elementele vectorului: ";
 for (int i=0; i < nr; i++) cin >> x[i];
}

void  afisare_vector(int nr, int a[10])
{
 cout << "Elementele vectorului sunt: ";
 for (int i = 0; i < nr; i++) cout << a[i]<<"  ";
 cout <<endl;
}
```

Exercițiu

Scrieți un program care citește o matrice pătratică `a` cu `n` x `n` elemente întregi și o matrice pătratică `b` cu `m` x `m` elemente întregi, apoi afișează pe ecran cele două matrice citite.

6.6. *Variabile globale și variabile locale*

Vom diferenția cele două categorii de variabile analizând următoarele caracteristici:

- poziția declarației variabilei;
- clasa de memorare (precizează zona de memorie în care este alocată variabila – în segmentul de date al programului, pe stivă etc.);
- durata de viață (precizează timpul în care variabila are alocată zonă de memorie);

- domeniul de vizibilitate (precizează zonele din program în care variabila respectivă este vizibilă, deci poate fi utilizată).

Variabilele globale

- *poziție*: sunt declarate în exteriorul oricărei funcții;
- *clasa de memorare*: li se alocă memorie în segmentul de date al programului, memoria alocată fiind automat inițializată cu `0`;
- *durata de viață*: memoria rămâne alocată până la sfârșitul execuției programului;
- *domeniu de vizibilitate*: sunt vizibile din momentul declarării până la sfârșitul programului, în toate modulele acestuia, chiar și în alte fișiere sursă[35], cu excepția cazurilor de omonimie.

Variabilele locale

- *poziție*: sunt declarate în blocul unei funcții;
- *clasa de memorare*: li se alocă memorie pe stivă, memoria alocată nefiind inițializată automat;
- *durata de viață*: memoria rămâne alocată până la sfârșitul execuției blocului în care este declarată variabila;
- *domeniu de vizibilitate*: sunt vizibile numai în blocul în care sunt declarate.

Regula de omonimie

Deoarece variabilele globale au altă clasă de memorare decât variabilele locale, este permis să declarăm o variabilă locală care să aibă același nume cu al unei variabile globale. O astfel de situație este denumită *omonimie*. Ce se întâmplă în astfel de cazuri?

Regula de omonimie: În interiorul blocului în care sunt declarate, variabilele locale *au prioritate* față de variabilele globale omonime.

Prin urmare, în cazul în care în interiorul unei funcții am declarat o variabilă locală cu același nume ca al unei variabile globale, variabila locală ascunde temporar (pe parcursul execuției blocului în care este declarată) variabila globală cu același nume.

Același lucru se întâmplă și atunci când numele unui parametru formal coincide cu al unei variabile globale: pe durata execuției funcției respective, parametrul are prioritate față de variabila globală omonimă.

---

35. În secțiunea „Proiecte" veți vedea că un program C/C++ poate fi constituit din mai multe fișiere sursă. Presupunând că am declarat o variabilă globală `v` în fișierul `f1.cpp` și dorim să o utilizăm și în fișierul `f2.cpp` din cadrul aceluiași program, este suficient să declarăm variabila `v` și în fișierul `f2.cpp`, precedând declarația ei de specificatorul extern. Aceasta este o indicație pentru compilator că variabila respectivă există într-un alt fișier sursă.

Operatorul de rezoluție

Limbajul C++ oferă programatorilor posibilitatea de a accesa în cadrul aceluiași bloc atât variabila locală, cât și variabila globală omonimă. Acest lucru este posibil cu ajutorul unui operator unar denumit operatorul de rezoluție:

```
::variabila
```

Această expresie desemnează variabila globală, și nu variabila locală omonimă.

Exemplu

```
int x;                                  //variabila globala
void f(void)
{ int x=5;                              //variabila locala
  cout <<"x local = " <<x << endl;
  cout <<"x global = "<<::x << endl;
}
```

În exemplul precedent am definit o variabilă globală `x`, apoi în cadrul funcției `f()` am definit o variabilă locală omonimă. În cadrul funcției, când ne referim la `x`, are prioritate variabila locală. Pentru a ne referi la variabila globală `x`, am utilizat construcția `::x`. Prin urmare, funcția `f()` va afișa:

```
x local = 5
x global = 0
```

6.7. *Specificatori de clasă de memorare*

În secțiunea precedentă am precizat clasa de memorare a variabilelor locale și a variabilelor globale. Acestea sunt clasele de memorare *implicite*. Dacă dorim, putem specifica în mod explicit clasa de memorare a unei variabile (fie ea locală sau globală), precizând înaintea declarației variabilei un cuvânt-cheie care reprezintă *specificatorul de clasă de memorare*.

Specificatorul `static`

În cazul variabilelor locale, indică faptul că variabila nu este memorată pe stivă, ci în segmentul de date. Prin urmare, ea este inițializată automat cu `0` și are zonă de memorie alocată de la prima execuție a blocului în care este declarată până la sfârșitul programului. Domeniul de vizibilitate nu se modifică (variabila rămâne vizibilă numai în interiorul blocului în care este declarată).

Avantajele utilizării variabilelor locale statice sunt multiple: se alocă o singură dată, nu la fiecare apel al funcției; își păstrează valorile între două apeluri consecutive ale funcției; domeniul de vizibilitate fiind local, nu există riscuri ca valoarea variabilei să fie alterată de alte funcții ale programului, ca în cazul variabilelor globale; dacă spațiul disponibil pe stivă devine o problemă, reprezintă o soluție de alocare a unor variabile locale.

În cazul variabilelor globale, indică faptul că variabila poate fi utilizată numai în cadrul fişierului sursă în care este declarată (deci nu mai are legătură externă, nu mai poate fi utilizată în alte fişiere sursă ale programului). Cu alte cuvinte, variabila devine locală fişierului sursă în care este declarată.

Exemplu

```
void f()
{ static int x=2;
  x++;
  cout << x << endl;}
```

Funcţia `f()` conţine variabila locală statică `x`. La primul apel al funcţiei, variabila `x` este iniţializată cu valoarea 2, apoi valoarea variabilei este incrementată şi se va afişa pe ecran valoarea 3.

La al doilea apel al funcţiei `f()`, variabila `x`, fiind statică, nu mai este nici alocată, nici iniţializată, ci îşi păstrează valoarea de la apelul precedent (3). Această valoare va fi incrementată, apoi se va afişa pe ecran valoarea 4.

Deci la fiecare apel al funcţiei `f()` se va afişa o altă valoare. Dacă nu am fi declarat variabila `x` utilizând specificatorul `static`, funcţia ar fi afişat întotdeauna valoarea 3.

Specificatorul **`auto`**

Se poate utiliza numai pentru variabile locale şi indică faptul că variabila este alocată pe stivă, ceea ce se întâmplă în mod implicit. Din acest motiv, acest specificator nu prea este utilizat.

Specificatorul **`register`**

Se utilizează pentru variabile locale şi reprezintă o indicaţie pentru compilator că accesul la această variabilă trebuie să fie cât mai rapid. Prin urmare, dacă este posibil (există regiştri disponibili), variabila va fi memorată în unul dintre regiştrii microprocesorului, nu în memoria RAM. Utilizarea acestui specificator poate fi utilă, de exemplu, pentru o variabilă contor într-o instrucţiune repetitivă.

6.8. *Parametrii funcţiei* `main()`

Funcţia `main()` are un statut special, deoarece ea este automat apelată la lansarea în execuţie a programului.

Până acum am descris funcţii `main()` fără parametri. Uneori este necesar ca sistemul de operare să comunice programului anumite informaţii. Mai exact, la lansarea în execuţie a programului din sistemul de operare în mod linie de comandă, i se pot transmite programului ca informaţii diferite opţiuni sau date iniţiale, ca argumente în linia de comandă. Argumentele din linia de comandă sunt succesiuni de caractere separate prin spaţii sau caractere TAB.

Pentru funcția `main()` se pot utiliza trei parametri:

```
main(int argc, char * argv[], char * env[]);
```

unde:

`argc` – indică numărul de argumente din linia de comandă (`argc > 0`);

`argv` – este un vector în care sunt reținute în ordine cele `argc` argumente din linia de comandă, ca șiruri de caractere (`argv[0]` este numele programului);

`env` – este un vector în care sunt reținute informații de mediu[36] ale sistemului de operare (de exemplu, prompterul sistemului, căi implicite de căutare), ultimul element al vectorului fiind `NULL`, pentru a marca sfârșitul listei.

Exemplu

Vom scrie un program care va primi ca argumente în linia de comandă numele utilizatorului și va afișa un mesaj de salut personalizat:

```
#include <iostream.h>
int main(int argc, char * argv[])
{if (argc==1)
     {cout << "Nu am primit numele utilizatorului";
      return 1;}
 cout <<"Salut, ";
 for (int i=1; i<argc; i++) cout << argv[i] <<' ';
 cout << "! \n";
 return 0;  }
```

Să presupunem că programul executabil se numește `buna.exe`. Lansând executabilul în mod linie de comandă astfel:

```
>buna Dan V. Ionescu
```

programul va prelua din linia de comandă patru argumente (`argc = 4`), valorile argumentelor fiind:

```
argv[]={"buna", "Dan", "V.", Ionescu"}
```

Efectul va fi:

```
Salut, Dan V. Ionescu!
```

Exercițiu

Scrieți un program care primește ca argument în linia de comandă numele unui fișier text și afișează pe ecran conținutul fișierului, pagină cu pagină. Mai exact, când ecranul este plin (au fost afișate 24 de linii), afișarea se întrerupe, se afișează pe ultima linie a ecranului mesajul `"Apasati o tasta pentru a continua"`, apoi se așteaptă ca utilizatorul să acționeze o tastă; după ce utilizatorul acționează o tastă, se afișează următoarea pagină ș.a.m.d.

36. Denumirea *env* provine din termenul *environment*, care înseamnă mediu.

6.9. *Caracteristici specifice funcțiilor C++*

Funcții inline

Limbajul C++ încurajează (uneori chiar impune) utilizarea funcțiilor „mici", care realizează un număr redus de operații specifice. Întrucât programatorii sunt preocupați de viteza de execuție a programelor lor, iar apelul unei funcții presupune executarea unor operații suplimentare (de exemplu, alocarea pe stivă a memoriei necesare apelului, copierea valorilor parametrilor, returnarea valorii calculate de funcție), în limbajul C++ a fost introdusă o categorie specială de funcții, denumite funcții *inline*.

Definiția unei funcții *inline* are același format general cu definiția unei funcții C uzuale, cu excepția faptului că este precedată de cuvântul rezervat `inline`.

Exemplu

```
inline max (int a, int b) {return a>b?a:b;}
```

Când compilatorul întâlnește o funcție *inline*, nu generează cod ca pentru o funcție obișnuită, ci înlocuiește apelul funcției cu secvența de instrucțiuni executată de funcția respectivă.

De obicei, pentru funcții mici, acest lucru nu duce la creșterea dimensiunii codului executabil (dacă luăm în calcul și operațiile suplimentare executate la apelul oricărei funcții). În plus, dacă o funcție *inline* este definită, dar nu este apelată niciodată, compilatorul nu generează cod pentru funcția respectivă, în timp ce pentru funcțiile uzuale, indiferent dacă funcția este sau nu apelată, codul este generat.

Funcții cu parametri impliciți

Pentru a corespunde unei varietăți cât mai mari de situații, funcțiile sunt proiectate de obicei cu un număr foarte mare de parametri, chiar dacă pentru unii parametri se utilizează frecvent la apel aceeași valoare.

Pentru ca funcțiile respective să fie mai ușor de utilizat, limbajul C++ a introdus o facilitate suplimentară, prin care programatorul poate declara în antetul funcției valori implicite.

Exemplu

```
void f(int a, int b=1, int c=2)
{cout<<a<<b<<c<<endl;}
```

Funcția `f()` are trei parametri, dintre care au doi au valori implicite (`b` are valoarea implicită `1`, iar `c` are valoarea implicită `2`).

La apelul funcției, se pot omite parametrii actuali corespunzători parametrilor formali cu valori implicite, în acest caz utilizându-se valorile implicite ale acestora. Pentru ca la apel corespondența parametru actual – parametru formal să se execute corect, întotdeauna parametrii cu valori implicite sunt ultimii declarați în lista de parametri a funcției.

De exemplu, funcția `f()` definită mai sus o putem apela astfel:

```
f(9);
```

În acest caz se utilizează valorile implicite ale parametrilor `b` și `c` și efectul este:

```
912
```

```
f(9,8);
```

În acest caz se utilizează valoarea implicită a parametrului `c` și efectul este:

```
982
```

```
f(9,8,7);
```

În acest caz nu se utilizează valorile implicite ale parametrilor și efectul este:

```
987
```

Un exemplu de funcție cu parametri impliciți pe care ați utilizat-o deja este funcția `getline()`, funcție membră a clasei `istream`:

Prototip:

```
istream & getline(signed char*, int, char = '\n');
```

Efect

Funcția citește caracterele de la intrare și le plasează în șirul descris de primul parametru. Operația se termină fie când s-au extras atâtea caractere câte specifică cel de-al doilea parametru, fie la întâlnirea marcajului de sfârșit, specificat de cel de-al treilea parametru.

Observați că cel de-al treilea parametru are ca valoare implicită caracterul *newline* (deci implicit citirea se termină la sfârșitul liniei).

Exercițiu

Descrieți o funcție de ordonare a unui vector. Funcția va primi ca parametri numărul de elemente din vector, vectorul și un parametru suplimentar care indică sensul de ordonare, cu valorile `0` – crescător, `1` – descrescător. În mod implicit, vectorul va fi ordonat crescător.

Supraîncărcarea funcțiilor

Cea mai puternică facilitate introdusă de limbajul C++ referitor la funcții constă în posibilitatea de supraîncărcare[37] a funcțiilor.

Supraîncărcarea oferă programatorului posibilitatea de a defini mai multe funcții cu același nume, dar care diferă prin numărul și/sau tipul parametrilor.

37. Termenul supraîncărcare este o traducere a termenului englezesc *overloading*. În unele cărți puteți întâlni și termenul supradefinire, cu aceeași semnificație.

Exemplu

Definim o funcție `suma()` care calculează suma a două numere întregi:

```
int suma (int a, int b)
{return a+b;}
```

Putem supraîncărca funcția `suma()`, astfel încât să calculeze suma elementelor unui vector:

```
int suma (int * a, int n)
{
  for (int i=0,s=0; i<n; i++) s+=a[i];
  return s;
}
```

Deși nu am discutat explicit despre aceasta, am folosit deja supraîncărcarea funcțiilor. De exemplu, funcția `get()`, funcție membră a clasei `istream`, este o funcție supraîncărcată, ea având următoarele prototipuri:

```
Forma 1: int get();
Forma 2: istream& get(signed char*, int len, char = '\n');
Forma 3: istream& get(unsigned char*, int len, char = '\n');
Forma 4: istream& get(unsigned char&);
Forma 5: istream& get(signed char&);
Forma 6: istream& get(streambuf&, char = '\n');
```

Compilatorul este cel care selectează la apel funcția adecvată în funcție de lista parametrilor actuali. În primul rând, se caută o corespondență exactă (o funcție pentru care tipurile parametrilor formali să coincidă cu cele ale parametrilor actuali ai apelului respectiv). Dacă nu este găsită o corespondență exactă, se fac conversii de tip implicite (de exemplu, tipul `float` este convertit la `double`; `char`, `unsigned char`, `unsigned int` sunt convertite la `int`).

Utilizarea funcțiilor supraîncărcate reprezintă unul dintre cele mai importante mecanisme ale programării orientate pe obiect.

Observație

Un caz particular de funcții sunt operatorii. Cei mai frecvent utilizați operatori supraîncărcați sunt operatorii de deplasare (<< și >>). În clasa `istream`, operatorul >> a fost supraîncărcat pentru a putea fi utilizat ca operator de citire, iar în cadrul clasei `ostream`, a fost supraîncărcat operatorul << pentru a putea fi utilizat ca operator de scriere. Când vom studia programarea orientată pe obiect, vom învăța despre modul de supraîncărcare a operatorilor.

Funcții șablon

În activitatea de programare apar frecvent situații în care trebuie să realizăm aceleași operații, dar pentru date diferite. De exemplu, trebuie să calculăm suma

unui șir de numere întregi, dar și suma unui șir de numere reale. Prin utilizarea funcțiilor supraîncărcate am putea defini două funcții de forma:

```
float suma (float *, int);
int   suma (int *, int);
```

Din păcate, chiar dacă instrucțiunile din blocul funcției coincid, ele trebuie scrise pentru fiecare funcție în parte.

Pentru astfel de situații, când funcțiile execută aceleași operații, dar pentru date de tipuri diferite, limbajul C++ oferă programatorilor posibilitatea de a defini funcții șablon[38].

O funcție șablon este un model cu ajutorul căruia compilatorul C++ generează funcții pentru diferite tipuri de date.

Definiția unei funcții șablon este precedată de cuvântul rezervat `template`, urmat de o listă de tipuri (un fel de parametri formali ai șablonului) încadrată între <>, fiecare tip specificat ca parametru fiind precedat de cuvântul rezervat `class`.

Exemplu

Definim o funcție șablon care să calculeze suma unui șir de numere, indiferent de tipul acestora:

```
template <class T>
T suma (T * a, int n)
{
  T s=0;
  for (int i=0; i<n; i++) s+=a[i];
  return s;
}
```

Observați că șablonul depinde de tipul `T`. La apelul unei funcții `suma()`, compilatorul va substitui `T` cu tipul corespunzător.

```
int   a[] = {1, 2, 3, 4, 5};
float b[] = {2.5, 2.3, 3.5, 1.2};
cout << suma(b, 4) << endl;
cout << suma(a, 5) << endl;
```

La primul apel al funcției `suma()`, `T` va fi înlocuit cu tipul `float`, la al doilea apel, `T` va fi înlocuit cu tipul `int`.

Funcțiile șablon se plasează, de obicei, într-un fișier antet care va fi inclus în program. Astfel, programele devin mai scurte, mai ușor de citit și de înțeles.

38. Termenul șablon este o traducere a termenului englezesc *template*.

Exerciții

1. Creați o funcție șablon care să ordoneze un vector, indiferent de tipul componentelor acestuia.
2. Creați o funcție șablon care să determine elementul maxim dintr-un vector, indiferent de tipul componentelor acestuia.
3. Creați un fișier antet care să conțină funcții șablon pentru operațiile cu stive și cozi implementate static.

6.10. *Proiecte*

O aplicație C/C++ nu este constituită în mod obligatoriu dintr-un singur fișier sursă. Definițiile funcțiilor din care este constituit programul se pot găsi în mai multe fișiere. Acest lucru permite elaborarea aplicațiilor în echipă (fiecare membru al echipei va salva funcțiile elaborate de el în fișiere sursă proprii), precum și reutilizarea anumitor funcții (dacă dorim ca într-o altă aplicație să utilizăm anumite funcții pe care le-am elaborat deja și se găsesc într-un fișier sursă, este suficient să includem în program fișierul sursă respectiv).

În final, după elaborarea tuturor fișierelor sursă, le vom asambla, prin crearea unui *proiect*.

În mediul de programare Borland C++ 3.1, *crearea* unui nou proiect sau *deschiderea* unui proiect existent se realizează cu ajutorul opțiunii *Open Project* din meniul *Project* al mediului de programare. Se va deschide o fereastră în care este vizualizată lista fișierelor sursă din care este constituit proiectul. Pentru *adăugarea* unui fișier sursă la proiectul deschis, se selectează opțiunea *Add Item* din meniul *Project*. *Eliminarea* din proiect a unui fișier sursă se realizează cu ajutorul opțiunii *Delete Item* din meniul *Project*. *Închiderea* unui proiect se realizează cu ajutorul opțiunii *Close Project* din meniul *Project*.

Exemplu

Să construim un proiect simplu, format din două fișiere sursă. Primul fișier sursă se numește `Radical.cpp` și conține definițiile a două funcții proprii de determinare a radicalului: `R2()` – pentru radicalul de ordin 2 și `R3()` – pentru radicalul de ordin 3.

Pentru calculul valorii radicalului de ordin doi vom folosi o metodă cunoscută sub numele de *algoritmul lui Hero*. Această metodă folosește pentru extragerea radicalului de ordin doi din numărul real pozitiv *a*, formula de recurență:

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right),\ n = 1, 2, \ldots$$

Demonstraţia matematică arată că, după un anumit număr de paşi[39], valoarea calculată a lui x_n dă o aproximaţie bună pentru $\sqrt{a}$.

Pentru radicalul de ordin trei vom folosi o formulă de recurenţă similară:

$$x_n = \frac{1}{3}(2x_{n-1} + \frac{a}{x_{n-1}^2}),\ n = 1, 2, \ldots$$

Pentru implementarea relaţiilor de recurenţă vom folosi două variabile, `x0` şi `x1`. În `x1` calculăm termenul curent, iar în `x0` reţinem termenul precedent.

Vom considera, de asemenea, că după zece paşi se obţine o aproximaţie suficient de bună pentru calculul valorilor respective cu şase zecimale exacte. Numărul de paşi poate fi determinat exact dacă se va considera drept condiţie de oprire:

$$|x1-x0| < 10^{-6}$$

ceea ce exprimă matematic faptul că primele şase zecimale ale celor două aproximaţii succesive sunt identice. În tabelul următor se poate observa faptul că în cazul calculării radicalului de ordin 2 s-a ajuns la şase zecimale exacte după cinci paşi, iar în cazul calculării radicalului de ordin 3 s-a ajuns la şase zecimale exacte după şapte paşi:

Paşi	x0	x1
1	8.000000	4.500000
2	4.500000	3.138889
3	3.138889	2.843781
4	2.843781	2.828469
5	2.828469	2.828427
6	2.828427	2.828427
7	2.828427	2.828427
8	2.828427	2.828427
9	2.828427	2.828427
10	2.828427	2.828427

Paşi	x0	x1
1	8.000000	5.375000
2	5.375000	3.675635
3	3.675635	2.647804
4	2.647804	2.145565
5	2.145565	2.009652
6	2.009652	2.000046
7	2.000046	2.000000
8	2.000000	2.000000
9	2.000000	2.000000
10	2.000000	2.000000

```
double R2(double a)
{double x0 = a, x1;
 for (int i = 1; i<=10; i++)
     { x1 = 0.5 * (x0 + a / x0);
       x0 = x1; }
 return x1;}
double R3(double a)
{
 double x0 = a, x1;
 for (int i = 1; i<=10; i++)
      { x1 = (1/3.0)*(2*x0 + a/(x0*x0));
        x0 = x1;    }
 return x1;
}
```

39. Vezi tabelul următor.

Al doilea fişier, denumit `F.cpp`, este constituit numai din funcţia `main()`, în cadrul căreia vom apela cele două funcţii:

```
#include <iostream.h>#include "radical.h"int main ()
{ double x;
  cout << "x= "; cin >> x;
  cout << "Radical de ordin 2= " << R2(x) << endl;
  cout << "Radical de ordin 3= " << R3(x) << endl;
  return 0;}
```

Observaţi că în fişierul `F.cpp` includem fişierul antet `Radical.h`, care conţine prototipurile celor două funcţii de determinare a radicalului:

```
double R2(double);
double R3(double);
```

După crearea celor două fişiere sursă şi a fişierului antet:

1. Selectăm opţiunea *Open Project* din meniul *Project*. Va apărea o fereastră de dialog în care vom introduce (în câmpul *Open Project File*) numele proiectului pe care îl creăm.
2. Selectăm opţiunea *Add Item* din meniul *Project*. Va apărea o fereastră de dialog, în cadrul căreia selectăm fişierul `F.cpp`, acţionăm butonul de comandă *Add*, apoi selectăm fişierul `Radical.cpp` şi acţionăm (în final) butonul de comandă *Done*.
3. Selectăm opţiunea *Make* sau *Build All* din meniul *Compile*, pentru a crea un program executabil. Programul executabil va fi un fişier cu numele proiectului şi extensia `.exe`.
4. Lansăm în execuţie programul, acţionând combinaţia de taste *CTRL+F9* sau selectând opţiunea *Run* din meniul *Run*.

6.11. *Aplicaţii*

Expresie

Să se scrie un program care să calculeze valoarea expresiei:

$$E=\begin{cases}\max(\min(|a|-3b,c-2|d|),\min(|a|+3b,c+2|d|)), & \text{daca } a+b>c-d\\ \min(\max(|a|-3b,c-2|d|),\max(|a|+3b,c+2|d|)), & \text{daca } a+b\le c-d\end{cases}$$

Soluţie

Studiind expresia, observăm că pentru a calcula valoarea expresiei se utilizează trei funcţii: calculul valorii absolute (`abs`), determinarea minimului dintre două

valori (min), determinarea maximului dintre două valori (max). Pentru fiecare dintre ele scriem câte o funcţie care returnează exact valoarea cerută. După definirea celor trei funcţii, calculul valorii expresiei devine extrem de simplu: se transcrie în limbaj expresia utilizând funcţiile definite:

```
#include <iostream.h>

float min(float x, float y);
float max(float x, float y);
float abs(float x);
int main ()
{ float a, b, c, d, e;
  cout << "a, b, c, d = "; cin >> a >> b >> c >> d;
  cout << (a+b > c-d)? max(min(abs(a)-3*b, c-2*abs(d)),
          min(abs(a)+3*b, c+2*abs(d))): min(max(abs(a)-3*b,
          c-2*abs(d)), max(abs(a)+3*b, c+2*abs(d)));
  return 0;}
float min(float x, float y)        //x,y - parametri formali
{
  if (y < x) return y;
  return x;
}
float max(float x, float y)
{
  if (y > x) return y;
  return x;
}
float abs(float x)
{
  if (x<0) return -x;
  return x;
}
```

Find & Replace

Fiind date un fişier text şi două şiruri de caractere (s1 şi s2), să se realizeze înlocuirea fiecărei apariţii a şirului s1 în fişierul dat cu şirul s2; fişierul iniţial va rămâne nemodificat, rezultatul prelucrării obţinându-se într-un alt fişier text.

Soluţie

Ideea este de a consulta fişierul sursă caracter cu caracter. Caracterele citite sunt plasate într-un şir auxiliar denumit buffer, de lungimea şirului s1. Dacă la un moment dat şirul buffer este egal cu şirul s1, în fişierul modificat se va scrie s2, iar din fişierul sursă se citesc caractere pentru reumplerea şirului buffer. Dacă şirul buffer nu este egal cu s1, se extrage primul caracter din buffer, se scrie în fişierul modificat, apoi se citeşte un nou caracter din fişierul sursă, care se introduce în buffer, la sfârşit.

```
#include <string.h>
#include <fstream.h>
ifstream f;
ofstream g;
char c, s1[200], s2[200], buffer[200], ns[100], nd[100];
int lbuffer, ls1, ls2;

void Citire()
{cout <<"Numele fisierului sursa:"; cin.getline(ns, 100);
cout<<"Numele fisierului modificat:";  cin.getline(nd,100);
cout << "Sirul cautat : "; cin.getline(s1, 200);
cout <<"Sirul cu care se inlocuieste: ";
cin.getline(s2,200); }

void UmpleBuffer()
{ int i = 1;
  char c[2];
  c[1] = buffer[0] = NULL;
  while ((i++ <= lbuffer) && !f.eof())
        { f.get(c[0]);
          strcat(buffer,c);  }
}

void Initializare()
{ f.open(ns); g.open(nd);
  lbuffer = ls1 = strlen(s1); ls2 = strlen(s2);
  //lbuffer - lungimea bufferului de intrare a caracterelor
  UmpleBuffer(); }

void Stanga(char * b, char & stanga, int lg)
{ //muta caracterele din buffer cu o pozitie la stanga
  stanga = b[0];
  for (int i = 1; i<lg; i++) b[i-1] = b[i]; }
int main()
{ Citire();
  Initializare();
  while (!f.eof())
       if (!strcmp(buffer,s1))
      //daca am gasit cuvantul s1, scriem s2 in destinatie}
           { g << s2;
             UmpleBuffer(); }          //reumplem bufferul
           else
          //scriem la destinatie primul caracter din buffer
           { Stanga(buffer, c, lbuffer);
             g << c;
             if (!f.eof())
                f.get(buffer[lbuffer-1]);
                             //citim un caracter in buffer
                else         //nu mai putem umple bufferul
                strncpy(buffer, buffer, lbuffer-1);
             }
```

```
    g << buffer;                    //scriem bufferul la destinatie
    f.close(); g.close();
    return 0; }
```

Cel mai mare divizor comun cu descompunere în factori primi

În capitolul precedent am studiat un algoritm pentru calculul celui mai mare divizor comun a două numere naturale (c.m.m.d.c.) utilizând algoritmul lui Euclid. La matematică, în clasele gimnaziale, am învățat și altă metodă de determinare a celui mai mare divizor comun: c.m.m.d.c. a două numere naturale se calculează astfel:

- se descompun cele două numere în factori primi;
- se calculează produsul factorilor primi comuni la puterea cea mai mică.

Să se scrie un program care calculează c.m.m.d.c. în acest mod.

Soluție

Vom considera cele două numere de tip `long int`. Avem nevoie, în primul rând, de doi vectori în care se vor reține factorii primi ai celor două numere și de încă doi vectori în care se vor reține puterile acestor factori. Trebuie, de asemenea, să determinăm numărul de divizori primi ai fiecărui număr.

Funcția `Descompune()` realizează descompunerea în factori primi a parametrului de intrare `X`, reținând factorii în vectorul `DivX`, iar puterea corespunzătoare fiecărui factor în vectorul `OrdX`. Funcția calculează și numărul de factori determinați, prin parametrul `i`.

În funcția de calcul și afișare a soluției (`Afiseaza()`) se parcurg simultan cei doi vectori și în cazul detectării unui factor comun se determină puterea cea mai mică, apoi se înmulțește c.m.m.d.c. cu factorul respectiv ridicat la puterea determinată. Deci, pentru calculul c.m.m.d.c. se folosesc două funcții: prima determină minimul a două valori, cealaltă calculează puterea naturală a unui număr.

```
#include <iostream.h>
#define Dim 20

typedef long Factori[Dim];
typedef int Multiplicitati[Dim];

Factori DivA, DivB;
Multiplicitati OrdA, OrdB;
int CatiA, CatiB;

void Descompune(long, int &, Factori, Multiplicitati);
long Putere (unsigned, unsigned);
long cmmdc();
```

```
int main()
{
  long int a, b;
  cout << "a = "; cin >> a;
  cout << "b = "; cin >> b;
  Descompune(a, CatiA, DivA, OrdA);
  Descompune(b, CatiB, DivB, OrdB);
  cout <<"c.m.m.d.c.("<<a<<", "<<b<<")= "<<cmmdc();
  return 0;}

void Descompune (long x, int & nx, Factori DivX,
                 Multiplicitati OrdX)
{ long  d = 2;
  nx = 0;
  while (x>1)
        {
         if (!(x % d))                    //am gasit un factor
            {
            DivX[++nx] = d;
            while (!(x % d))      //calculez multiplicitatea
                  { x/=d;
                    OrdX[nx]++; }
            }
          d++;
        }
}
long Putere (unsigned x, unsigned y)
//calculeaza x la puterea y
{ unsigned i;
  long P=1;
  for (i = 1; i <= y; i++) P *= x;
  return P; }
long cmmdc()
{
int i = 1, j = 1;
long c = 1;
while ((i <= CatiA) && (j <= CatiB))
      if (DivA[i] == DivB[j])     //am gasit un factor comun
       {          //determin puterea minima apoi calculez
        c*=Putere(DivA[i],OrdA[i]>OrdB[j]?OrdA[i]:OrdB[j]);
        i++; j++;                 //trec la factorii urmatori
        }
        else
        if (DivA[i] < DivB[j]) i++;
           else j++;
return c;
}
```

Incluziune

Se consideră un fișier text cu numele `prim.txt` care conține două linii. Pe prima linie se află elementele mulțimii `A`, iar pe a doua linie sunt scrise elementele mulțimii `B`. Elementele celor două mulțimi sunt din mulțimea

```
['a'..'z','A'..'Z','0'..'9']
```

și sunt separate între ele prin câte un spațiu. Scrieți un program care scrie pe prima linie a fișierului de tip text `doi.txt` cuvântul `'DA'`, dacă mulțimea `A` este inclusă în `B`, respectiv `'NU'` în caz contrar.

(Bacalaureat, iunie 1999)

Soluție

Vom reprezenta o mulțime de caractere prin vectorul caracteristic. Mai exact, o mulțime de caractere va fi un vector cu 256 de componente care pot avea valorile `0` sau `1`. Dacă pe poziția `i` în vectorul caracteristic se găsește valoarea 1, caracterul cu codul ASCII `i` aparține mulțimii. Dacă pe poziția `i` în vectorul caracteristic se găsește valoarea `0`, caracterul cu codul ASCII `i` nu aparține mulțimii.

Pentru a citi o mulțime utilizăm funcția numită `Citeste_Multimea()`, care citește caracterele de pe linia curentă a fișierului de intrare (până la întâlnirea caracterului *newline*) și le reunește la mulțimea specificată ca parametru. Parametrul este obligatoriu, deoarece în programul principal apelăm această funcție de două ori: o dată pentru a citi mulțimea `A` și încă o dată pentru a citi mulțimea `B`. Parametrul trebuie să fie de intrare/ieșire (transmis prin referință). Funcția `Compara()` realizează compararea celor două mulțimi construite. În cadrul funcției `Compara()` apelăm funcția `Inclus()`, care testează dacă mulțimea specificată de primul parametru este inclusă în mulțimea specificată de cel de-al doilea parametru.

```
#include <fstream.h>

typedef int Multime[256];
Multime A, B;
ifstream p("prim.txt");

void Citeste_Multimea(Multime X);
void Compara(Multime X, Multime Y);
int Inclus(Multime X, Multime Y);

int main ()
{
  Citeste_Multimea(A);              //construiesc multimea A
  Citeste_Multimea(B);              //construiesc multimea B
  p.close();
  Compara(A, B);                       //verific incluziunea
  return 0;           }
```

```
void Citeste_Multimea(Multime X)
{ char c;
  do
     { c=p.get();        //citesc un caracter
       X[c]=1;   }       //adaug caracterul c la multime
  while (c!='\n');
  X[' '] = X[10] = 0; //extrag caracterele spatiu si LF
}

void Compara(Multime X, Multime Y)
{ ofstream d("doi.txt");
  if (Inclus(X, Y))           //daca X este inclus in Y...
     d << "DA"<<endl;
     else
     d << "NU"<<endl;
  d.close();              }

int Inclus(Multime X, Multime Y)
{//intoarce 1 daca X inclus in Y si 0 altfel
 int i;
 for (i='0'; i<='9'; i++)
     if (X[i]>Y[i]) return 0;
 for (i='A'; i<='Z'; i++)
     if (X[i]>Y[i]) return 0;
 for (i='a'; i<='z'; i++)
     if (X[i]>Y[i]) return 0;
 return 1;                  }
```

Observație

Funcția `Citeste_Multimea()` trebuie să modifice valoarea parametrului său. Parametrul său este de tip `Multime`, deci este un vector. Prin urmare, la apelul funcției, pe stivă este memorat un pointer care are ca valoare adresa primului element din vector.

Operații cu numere naturale mari

Să considerăm numere naturale mari (cu maximum `200` de cifre). Nici unul dintre tipurile limbajului nu permite lucrul cu astfel de numere. Prin urmare, trebuie să implementăm propriul nostru tip pentru numere mari și operațiile uzuale cu numere mari (adunare, scădere, înmulțire, împărțire, comparare).

Soluție

Vom reprezenta un număr natural mare ca un vector în care reținem în ordine cifrele sale, începând cu unitățile (astfel, pe poziția `0` se va afla cifra unităților, pe poziția `1`, cifra zecilor ș.a.m.d.). Observați că indicele din vector al cifrei coincide cu puterea bazei corespunzătoare cifrei.

```
#define Lg 200
typedef char NrMare[Lg];
```

Citirea unui număr mare

Pentru a citi un număr mare vom citi de la tastatură numărul într-un șir de caractere, apoi vom transforma caracterele cifră în numere și le vom memora în ordine inversă în vectorul cu cifrele numărului mare. Pentru a ușura calculele, vom completa zona de memorie alocată numărului mare (celelalte componente ale vectorului de cifre) cu 0. Funcția `citire()` va avea doi parametri: vectorul în care reținem cifrele numărului mare citit (`NrMare x`) și numărul de cifre ale numărului mare (`int & nx`). Observați că am transmis parametrul `nx` prin referință (pentru ca funcția `citire()` să poată modifica valoarea acestui parametru, valoarea rămânând modificată și după apel). Parametrul `x` nu este necesar să fie transmis prin referință, deoarece `x` este numele unui vector, deci este un pointer constant la primul element din vector.

```
void citire (NrMare x, int &nx)
{ int i;
  char s[Lg+1];
  cin>>s;
  nx=strlen(s); //determin numarul de cifre
  for (i=nx-1; i>=0; i--) x[nx-i-1]=s[i]-'0';
  for (i=nx; i<Lg; i++) x[i]=0;
}
```

Afișarea unui număr mare

Vom parcurge vectorul de cifre de la sfârșit către început, afișând succesiv cifrele. Funcția de afișare va avea doi parametri (vectorul în care reținem cifrele numărului mare citit și numărul de cifre ale numărului mare).

```
void afisare (NrMare x, int nx)
{for (int i=nx-1; i>=0; i--) cout <<(int)x[i];
 cout<<endl; }
```

Suma a două numere mari

Pentru a calcula suma a două numere mari vom parcurge simultan cele două numere, începând de la unități. Vom aduna cele două cifre de pe poziții corespondente și vom lua în calcul și eventuala cifră de transport care s-a obținut de la adunarea precedentă. Reținem în sumă cifra obținută prin adunare și recalculăm transportul. Suma ar putea avea o cifră în plus față de cel mai mare dintre numere (dacă la sfârșitul adunării cifra de transport este nenulă).

Funcția `suma()` va avea ca parametri cele două numere mari care se adună și numărul mare rezultat (vectorul de cifre și numărul de cifre pentru fiecare). Observați că parametrul `ns` (numărul de cifre ale numărului mare obținut în urma adunării) a fost transmis prin referință.

```
void suma (NrMare a, int na, NrMare b, int nb,
           NrMare s, int & ns)                    //s=a+b
{int i, cifra, t=0, max;
```

```
//completam numarul cel mai mic cu zerouri nesemnificative
if (na<nb) {max=nb; for (i=na; i<nb; i++) a[i]=0;}
    else   {max=na; for (i=nb; i<na; i++) b[i]=0;}
for (i=0; i<max; i++)
     {cifra=a[i]+b[i]+t;
      s[i]=cifra%10;
      t=cifra/10;         }
if (t) s[i++]=t;
ns=i; }
```

Diferența a două numere mari

Vom presupune că descăzutul este mai mare sau egal cu scăzătorul. Pentru a calcula diferența a două numere mari vom parcurge descăzutul începând de la unități. Vom scădea din cifra curentă a descăzutului cifra curentă a scăzătorului și vom lua în calcul și eventuala cifră de transport, obținută de la scăderea precedentă. Dacă rezultatul este negativ, împrumutăm `10` de pe poziția următoare (este posibil, dacă descăzutul este mai mare decât scăzătorul), și în acest caz cifra de transport devine `-1`. Diferența ar putea avea mai puține cifre decât descăzutul, astfel încât la sfârșit calculăm numărul de cifre din diferență, ignorând zerourile nesemnificative.

```
void dif (NrMare a, int na, NrMare b, int nb,
          NrMare d, int & nd)                    //d=a-b
{int i, t=0;
 for (i=0; i<na; i++)
     {d[i]=a[i]-b[i]+t;
      if (d[i]<0)
         {d[i]+=10; t=-1;}
         else t=0;
     }
 i--;
 while (i && !d[i]) i--;
 nd=i+1; }
```

Produsul a două numere mari

Vom inițializa vectorul de cifre ale produsului cu `0`. Vom înmulți fiecare cifră a înmulțitorului (`b`) cu deînmulțitul (`a`) și vom aduna acest produs parțial obținut la produs (ale cărui cifre sunt memorate în vectorul `p`). Observați că atunci când înmulțim cifra `b[j]` cu `a`, produsul parțial obținut este înmulțit cu 10^j (adică cifrele sunt deplasate cu `j` poziții la dreapta).

```
void produs (NrMare a, int na, NrMare b, int nb,
             NrMare p, int & np)                 //p=a*b
{int i, j, t, cifra;
 for (i=0; i<Lg; i++) p[i]=0;
 for (i=0; i<nb; i++)
     {for (t=j=0; j<na; j++)
          {cifra=p[i+j]+a[j]*b[i]+t;
           p[i+j]=cifra%10;
           t=cifra/10;}
```

```
      if (t) p[i+j]=t;      }
 np=na+nb-1;
 if (p[np]) np++; }
```

Compararea a două numere mari

Dacă numărul de cifre din a este mai mic decât numărul de cifre din b, atunci a<b (și invers). Dacă numerele a și b au același număr de cifre, atunci parcurgem cele două numere simultan, începând de la cifra cea mai semnificativă până la întâlnirea a două cifre diferite. Dacă am întâlnit două cifre diferite, ele determină ordinea dintre a și b; în caz contrar, numerele sunt egale.

```
int compara(NrMare a, int na, NrMare b, int nb)
/*functia returneaza -1 daca a<b; 0 daca a=b; 1 daca a>b */
{int i;
 if (na<nb) return -1;
 if (nb<na) return 1;
 for (i=na-1; i>=0 && a[i]==b[i]; i--);
 if (i<0) return 0;
 if (a[i]<b[i]) return -1;
 return 1;}
```

Înmulțirea unui număr mare cu o putere a lui 10

Pentru a înmulți un număr mare cu 10^{nr} trebuie să deplasăm cifrele numărului cu nr poziții la dreapta și să completăm în stânga cu zerouri.

```
void p10(NrMare x, int &nx, int nr)
{// inmulteste numarul mare x cu 10^nr
 int i;
 for (i=nx-1; i>=0; i--) x[i+nr]=x[i];
 for (i=0; i<nr; i++) x[i]=0;
 nx+=nr;   }
```

Împărțirea a două numere mari

Dacă cele două numere mari sunt comparabile ca dimensiune, putem simula împărțirea prin scăderi repetate; în caz contrar, această metodă este ineficientă. Vom simula algoritmul de împărțire învățat la matematică:

```
void div(NrMare a, int na, NrMare b, int nb,
         NrMare c, int & nc, NrMare r,int &r)
{// imparte a la b; c este catul; r este restul
  int i;
  nr=0; nc=na;
  for (i=na-1; i>=0; i--)
    { p10(r,nr,1); r[0]=a[i];
      c[i]=0;
      while (compara(b,nb,r,nr)!=1)
            {c[i]++;
             dif(r,nr,b,nb,r,nr);  }
    }
```

```
while (!c[nc-1] && nc>1) nc--;
while (!r[nr-1] && nr>1) nr--; }
```

Secvență regulată

Să considerăm secvențe constituite numai din paranteze rotunde și paranteze pătrate, adică din caracterele `()[]`. Prin definiție, o secvență de paranteze este regulată dacă se obține aplicând următoarele reguli:

1. `()` și `[]` sunt secvențe regulate;
2. Dacă `A` este regulată, atunci `(A)` și `[A]` sunt secvențe regulate;
3. Dacă `A` și `B` sunt secvențe regulate, atunci `AB` este secvență regulată.

De exemplu, `()()[]` și `[(())][]` sunt secvențe regulate, în timp ce `][` sau `[(]` sau `([)]` nu sunt secvențe regulate.

Se consideră o secvență de paranteze. La fiecare pas se inserează o paranteză (rotundă sau pătrată, deschisă sau închisă) la începutul sau la sfârșitul secvenței. Scrieți un program care să determine, după fiecare pas, lungimea celei mai scurte subsecvențe regulate (formată din caractere consecutive ale secvenței) care conține paranteza inserată la pasul respectiv.

Date de intrare

Fișierul de intrare `secvreg.in` conține pe prima linie secvența inițială de paranteze (de lungime maximă `10000`). Pe cea de-a doua linie a fișierului de intrare se află un număr natural `N` care reprezintă numărul de pași (`N≤10000`). Fiecare dintre următoarele `N` linii conține un număr natural `A` și un caracter `C` separate printr-un singur spațiu. Dacă `A` este `0`, atunci caracterul `C` este inserat la începutul secvenței; dacă `A` este `1`, atunci caracterul `C` este inserat la sfârșitul secvenței.

Date de ieșire

Fișierul de ieșire `secvreg.out` conține `N` linii. Pe cea de a `i`-a linie va fi afișată lungimea celei mai scurte subsecvențe regulate (formată din caractere consecutive ale secvenței) care conține paranteza inserată la pasul `i`. Dacă nu există o astfel de subsecvență, pe linia respectivă veți afișa `0`.

Exemple

secvreg.in	secvreg.out	secvreg.in	secvreg.out
`[])`	`0`	`(]`	`0`
`3`	`2`	`3`	`0`
`0 )`	`6`	`1 )`	`2`
`0 (`		`0 )`	
`0 (`		`0 (`	

(.campion, 2004)

Soluție

Numim coadă cu dublă prioritate (*dequeue*) o structură de date abstractă care suportă următoarele operații:

- inserarea unui element la sfârșit;
- inserarea unui element la început;
- extragerea elementului de la sfârșit;
- extragerea elementului de la început;
- accesarea elementului de la început;
- accesarea elementului de la sfârșit.

Vom utiliza două cozi cu dublă prioritate:

1. Coada `a` cu elemente de tip `char`, în care inserăm succesiv caracterele din șirul inițial și ulterior cele specificate în operațiile din fișierul de intrare.
2. Coada `b` cu elemente de tip `int`, cu următoarea semnificație: `b[i]` = lungimea secvenței regulate care începe cu poziția `i`.

Implementam cozile cu dublă prioritate `a` și `b` ca doi vectori. Pentru a putea realiza operații și la începutul, și la sfârșitul vectorilor, am dublat spațiul de memorie necesar pentru fiecare dintre cei doi vectori și am început inserarea elementelor de la mijloc.

Inițial, inserăm în ordine toate caracterele șirului citit în coada `a`. Pentru a păstra ordinea, fiecare nou caracter va fi inserat la sfârșitul cozii. Când inserăm o paranteză închisă la sfârșitul cozii, iar pe prima poziție în coadă este paranteza deschisă corespunzătoare, deducem că se formează o secvență regulată cu lungimea cu 2 mai mare decât cea deja existentă pe această poziție. Prin urmare, reținem în vectorul `b` această lungime și eliminăm din coada `a` cele două paranteze (practic, fiecare secvență regulată formată este eliminată din `a`, dar în `b` reținem lungimile secvențelor formate). Când inserăm la începutul cozii `a` o paranteză deschisă, iar la sfârșitul cozii `a` există paranteza închisă corespunzătoare, se formează o secvență regulată cu lungimea cu 2 mai mare și executăm aceleași operații ca în cazul precedent.

```
#include <stdio.h>
#define LgMax 10000
#define InFile "secvreg.in"
#define OutFile "secvreg.out"
typedef int CoadaI[LgMax];
typedef char CoadaC[LgMax];
char pereche[100];
CoadaC a;
CoadaI b;
/* inca = inceputul cozii a; sfa = sfarsitul cozii a;
   initial coada a este vida */
int inca=LgMax/2, sfa=LgMax/2-1;
/* incb = inceputul cozii b; sfb = sfarsitul cozii b;
   initial coada b este vida */
int incb=LgMax/2, sfb=LgMax/2-1;
char s[LgMax];

int deschis(char c)
{return c=='(' || c=='[';}
```

```
int la_sfarsit(char c)
/* functia insereaza caracterul c la sfarsitul cozii a si
returneaza lungimea secventei regulate care contine
caracterul inserat */
{ int ret;
  if ( inca<=sfa && !deschis(c) && a[sfa] == pereche[c] )
     {ret=b[sfb]+2;/* retin lungimea secventei regulate*/
      sfa--; sfb--; /* elimin secventa regulata formata */
     b[sfb]+=ret; /*memorez in b lungimea noii secvente*/
     return ret;      }
  a[++sfa]=c; /*adaug la sfarsitul cozii a caracterul c */
  b[++sfb]=0;
  /* cu c se formeaza o secventa regulata de lungime 0 */
  return 0; }

int la_inceput(char c)
/* functia insereaza caracterul c la inceputul cozii a si
returneaza lungimea secventei regulate care contine
caracterul inserat */
{ int ret;
  if (inca<=sfa && deschis(c) && a[inca] == pereche[c] )
     {ret=b[incb]+2; /* retin lungimea secventei regulate*/
     inca++; incb++; /* elimin secventa regulata formata */
     b[incb]+=ret; /* memorez in b lungimea noi secvente */
     return ret;      }
  a[--inca]=c ; /*adaug la inceputul cozii a caracterul c*/
  b[--incb]=0 ;
  /* cu c se formeaza o secventa regulata de lungime 0 */
  return 0; }
int main(void)
{ char * c, ch;
  int n, di, i;
  FILE *fin, *fout;
  fin=fopen(InFile, "rt");
  fscanf(fin,"%s",s);
  fout=fopen(OutFile,"wt");
  pereche['('] = ')'; pereche[')'] = '(';
  pereche['['] = ']'; pereche[']'] = '[';
  /* deocamdata elementele din coada a formeaza o secventa
     regulata de lungime 0 */
  b[++sfb]=0;
  /* inserez succesiv caracterele din sirul citit la
     sfarsitul cozii a */
  for (c=s; *c; ++c ) la_sfarsit( *c );
  /* citesc numarul de operatii */
  fscanf(fin, "%d", &n );
  for (i = 0; i < n; i++ )
      {/* citesc succesiv operatiile */
      fscanf(fin, "%d %c", &di, &ch );
```

```
      if (di) fprintf(fout,"%d\n", la_sfarsit(ch));
         else fprintf(fout,"%d\n", la_inceput(ch));
      }
  fclose(fin);
  fclose(fout);
  return 0; }
```

6.12. *Probleme propuse*

1. Care dintre următoarele propoziții sunt adevărate și care sunt false? Justificați!
   a. Parametrii formali ai unei funcții sunt cei care intervin la apelul funcției.
   b. La apelul unei funcții se specifică numele și tipul parametrilor actuali.
   c. Parametrii actuali trebuie să corespundă cu parametrii formali ca număr, ordine și tip.
   d. La apelul unei funcții, valorile parametrilor actuali sunt copiate pe stivă.
   e. Funcțiile trebuie să aibă cel puțin un parametru formal.

2. Funcția `f` primește ca parametru un număr real și trebuie să determine numărul de cifre de la partea întreagă și numărul de cifre de la partea zecimală a numărului. Care sunt prototipuri corecte pentru funcția `f`? Justificați răspunsul!
   a. `void f(float a, int nr1, int nr2);`
   b. `void f(float & a, int nr1, int nr2);`
   c. `void f(float, int *, int *);`
   d. `void f(float ; int & ; int &);`
   e. `int f(float, int &);`

3. Să considerăm funcția `Sum(int y)` care returnează suma elementelor plasate într-un vector până la poziția `y` (inclusiv). Care dintre următoarele expresii are valoarea diferită de `0` dacă și numai dacă elementul de pe poziția `i` din vector este strict pozitiv?
   a. `Sum(i)>0 && Sum(i-1)<=0`
   b. `Sum(i)-Sum(i-1)>0`
   c. `Sum(i)>0 && Sum(i+1)<=0`
   d. `Sum(i)>0 && Sum(i-1)>0 && Sum(i+1)>0`

4. Alegeți propozițiile adevărate. Justificați alegerea!
   a. Programul următor va afișa `0363153159`.
   b. Programul următor conține erori de sintaxă.
   c. Programul următor va afișa `0303163169`.
   d. Programul următor va afișa `036315399`.

```
#include <iostream.h>
int x, y;
void F(int, int &);
void main ()
```

```
{ F(x, y); cout << x << y;
  F(y, x); cout << x << y;
  F(x, x); cout << x << y;
  F(y, y); cout << x << y; }
void F(int a, int & b)
{  a+=3;  b+=a; }
```

5. Alegeți varianta corectă. Justificați alegerea!

a. Programul afișează:
```
4122
4412
8244
8124
```

b. Programul afișează:
```
4422
4112
8244
8124
```

c. Programul afișează:
```
4422
4112
4244
8128
```

d. Programul conține erori de sintaxă.

```
#include <iostream.h>
int a=1, b=1, c=1, d=1;
void P(int &, int);
void main ()
{ P(a, b);   cout << a << b << c << d << endl;
  P(c, d);   cout << a << b << c << d << endl; }
void P(int & b, int c)
{  a *= 2; b *= 2; c *= 2; d *= 2;
   cout << a << b << c << d << endl; }
```

6. Funcția următoare trebuie să calculeze numărul de cifre egale cu `0` din scrierea numărului întreg `n`, primit ca parametru. Care sunt erorile din această funcție?

```
int nr0(int n)
{int nr;
 while (nr)
      {if (!nr%10) nr++;
       nr/=10; }
}
```

7. Ce va afișa pe ecran următorul program?

```
#include <iostream.h>
int a=5, b=10, c=20;
void p(int a, int & b)
{a++; b++; c++;
 cout<<a<<b<<c;  }

int main()
{p(a, b); cout<<a<<b<<c;
 p(c, c); cout<<a<<b<<c;
 return 0; }
```

8. Alegeți afirmațiile adevărate. Justificați alegerea!

a. Programul următor conține erori de sintaxă.

b. Programul următor afișează `1.0 2.0`.

c. Programul următor afișează `0 2`.

d. Programul următor afișează `1.0 0.0`.

```
#include <iostream.h>
float a[2];
int i;
void P(int & k, float & x)
{  k = 1; x = 0;  }
int main(void)
{ a[0] = 1; a[1] = 2;
  for (i = 0; i < 2; i++)
        {P(i, a[i]);
        cout << a[0] << " " << a[1] << endl; \}
  return 0;      }
```

9. Ce va afișa pe ecran următorul program?

```
#include <iostream.h>
int a[5]={1,2,3,4,5}, n=2;
int P(int, int &, int);
void main()
{ int i;
  P(n, a[1], a[2]);
  for (i=0; i<=n; i++) cout<<a[i];
  P(n, a[0], a[1]);
  for (i=0; i<=n; i++) cout<<a[i];
  return 0; }
void P(int n, int &x, int y)
{n++; x++; y++; }
```

10. Cum trebuie declarat antetul subprogramului `f` astfel încât programul următor să afișeze pe ecran valoarea `147`?

```
#include <iostream.h>
void f...
{a++; b+=a; c+=b;}
void main()
{int a=1, b=2, c=3;
 f(a,b,c);
 cout<<a<<b<<c;}
```

a. `void f(int, int , int );`

b. `void f(int & a, int b, int c);`

c. `void f(int a, int & b , int & c );`

d. `void f(int &a, int &b, int & c);`

e. `void f(int &a , int b, int & c );`

f. Indiferent de antet, aceste valori nu pot fi obținute.

11. Fie a, b, c, d numere reale citite de la tastatură. Să se calculeze valoarea expresiei:

$$E(a,\ b,\ c,\ d) = \begin{cases} \max(|a|-3, |b|+6), & \text{daca } c+2d<=3 \\ \min(|a|-3, |b|+6), & \text{daca } 3<c+2d<7 \\ \min(|a|+|b|, 18), & \text{daca } c+2d\geq 7 \end{cases}$$

12. Fie a şi b două numere complexe citite de la tastatură. Să se evalueze expresia pentru o valoare dată a variabilei complexe z:

$$E(z) = \begin{cases} 3z^2-6az+b, & \text{daca } |z|<1 \\ 6az-b, & \text{daca } |z|>1 \end{cases}$$

13. Scrieţi o funcţie care să determine raza cercului circumscris unui triunghi dat prin coordonatele vârfurilor.

14. Scrieţi o funcţie care să elimine zerourile dintr-un vector de numere întregi transmis ca parametru, fără a modifica ordinea elementelor nenule din vector. Descrieţi şi o funcţie şablon care să realizeze acelaşi lucru, indiferent de tipul elementelor vectorului.

15. Fie două numere naturale n şi m şi o funcţie $f:\{1,\ldots,n\}\to\{1,\ldots,m\}$. Scrieţi câte o funcţie care să testeze injectivitatea, surjectivitatea, respectiv bijectivitatea funcţiei.

16. Concepeţi câte o funcţie pentru următoarele operaţii cu şiruri de caractere:

a. Inserarea unui şir de caractere într-un alt şir de caractere; se transmit ca parametri cele două şiruri şi poziţia i în care se face inserarea.

b. Ştergerea unei secvenţe de caractere dintr-un şir; şirul de caractere, poziţia i de la care începe secvenţa de caractere ce va fi ştearsă şi numărul de caractere care se şterg sunt specificate ca parametri.

17. *Operaţii cu mulţimi*

Fie n un număr natural (n<1024). O metodă eficientă de a reprezenta o submulţime S a mulţimii {0, 1, ..., n} este vectorul caracteristic. Mai exact, asociem fiecărui element x din mulţimea {0, 1, ..., n} un bit. Bitul corespunzător elementului x are valoarea 1, dacă $x\in S$, respectiv valoarea 0, dacă $x\notin S$. Descrieţi funcţii care să implementeze operaţiile elementare cu mulţimi (reuniune, intersecţie, diferenţă, incluziune, apartenenţă) folosind o astfel de reprezentare.

18. *Ciurul lui Eratostene*

Creaţi un proiect în care utilizaţi fişierul cu funcţiile de lucru cu mulţimi elaborat la problema precedentă. Proiectul va avea ca scop generarea tuturor numerelor prime mai mici decât 1024, folosind metoda ciurului lui *Eratostene*.

19. *Traducere*

Să se scrie un program care realizează traducerea „cuvânt cu cuvânt" a unui text pe baza unui dicționar. Textul care trebuie tradus, traducerea, precum și dicționarul sunt fișiere text ale căror nume se citesc de la tastatură. Cuvintele de pe o linie sunt separate prin spații și/sau semne de punctuație (',', '.', ';', ':') și au lungimea maximă de 20 de caractere. Lungimea maximă a unei linii din fișier este de 100 de caractere. În dicționar, fiecare rând are forma `cuvant=traducere`.

20. *Jocul vieții și al morții*

Acest joc a fost propus de matematicianul John Conway și poate fi descris după cum urmează.

Un habitat este un tablou bidimensional cu `n` linii și `m` coloane; în fiecare element al acestui tablou poate fi o celulă. Vecinătatea unei poziții este constituită din cele 8 poziții din jurul ei. Numărul de vecini ai unei celule este numărul de celule care se află în vecinătatea ei. Populația evoluează în pași discreți: la momentul `t+1`, configurația populației este determinată doar de configurația de la momentul `t`. Legile după care evoluează populația sunt:

- dacă o celulă are 0 sau 1 vecini, ea moare prin izolare;
- dacă o celulă are mai mult de 3 vecini, ea moare prin sufocare;
- dacă în vecinătatea unei poziții goale se află 3 celule, atunci în acea poziție se naște o celulă;
- în toate celelalte cazuri nu se înregistrează vreo schimbare.

Să se scrie un program care simulează acest joc, vizualizând pe ecran, în mod text, fiecare pas.

21. *Apariție*

Fie `x` un număr natural (`x<MAXLONG`). Se cere să se determine numărul de apariții ale lui `x` ca subsecvență de cifre în numerele din intervalul dat `[a,b]` (`a` și `b` fiind numere naturale, `a<b<MAXLONG`). De exemplu, pentru `a=1`, `b=60`, `x=5` veți afișa `16`.

22. *Comparație*

Scrieți un program care realizează o comparație între metodele de sortare învățate până acum (*bubblesort*, sortare prin selecție, sortare prin inserție). Realizați o reprezentare grafică animată, precum și estimarea timpului de sortare pentru fiecare dintre cele trei metode.

23. *Baze ascunse*

Se dau `n` numere naturale, x_1, x_2, ..., x_n, de câte cel mult `9` cifre. Fiecare număr x_i este reprezentat într-o bază b_i ($2 \le b_i \le 10$), care nu este cunoscută. Să se determine pentru fiecare număr x_i o bază b_i astfel încât intervalul de valori `[a,b]` în care sunt cuprinse toate cele `n` numere, scrise în baza `10`, să fie de lungime minimă. Prin lungimea unui interval se înțelege diferența dintre capetele intervalului în valoare absolută. De exemplu, pentru numerele `35`, `27`, `100`, `13`, `19`, în fișierul de ieșire veți afișa:

```
35 in baza 6
27 in baza 8
100 in baza 4
13 in baza 10
19 in baza 10
a=13  b=23
```

(Olimpiada Națională de Informatică, Timișoara, 1997)

24. *Petice*

La grădiniță, Vasilică a învățat să coasă. Doamna educatoare i-a dat petice dreptunghiulare de material (de dimensiuni numere naturale), de culori distincte, și o bucată de pânză albă, de formă dreptunghiulară (de dimensiuni, de asemenea, numere naturale), pe care Vasilică urma să coasă peticele, așezându-le laturile paralel cu laturile bucății de pânză. Doamna educatoare i-a spus că are voie să coasă un petic peste altul, cu condiția ca întotdeauna peticul de deasupra să acopere cel mult un colț al oricărui petic de dedesubt. După ce și-a terminat opera, Vasilică o duce doamnei educatoare și o întreabă care este culoarea ce ocupă cea mai mare suprafață și în ce ordine a cusut el peticele. Scrieți un program care să o ajute pe doamna educatoare să îi răspundă lui Vasilică.

Din fișierul de intrare `petice.in` se citesc de pe prima linie două numere naturale `n` și `m` reprezentând lățimea și, respectiv, lungimea bucății de pânză albă (`n, m<100`). Urmează `n` linii, pe fiecare linie fiind specificate `m` caractere. Al `j`-lea caracter de pe linia `i+1` reprezintă culoarea punctului de coordonate `i, j` de pe pânză, culorile fiind codificate cu litere mici ale alfabetului englez, 'a' reprezentând alb, culoarea pânzei.

Fișierul de ieșire `petice.out` va conține pe prima linie un caracter `cmax` reprezentând culoarea ce ocupă cea mai mare suprafață, iar pe cea de-a doua linie $c_1c_2 \ldots c_n$ = culorile peticelor, în ordinea în care au fost cusute pe pânză.

25. *Pepsi*

Într-o cameră există `M` sticle identice, care conțin fiecare câte `K` dl de Pepsi. Sticlele sunt numerotate distinct de la `1` la `M`. Fiind cald, s-a format o coadă din `N` persoane însetate. Persoanele sunt numerotate de la `1` la `N`, în ordinea în care s-au așezat la coadă (deci prima persoana de la coadă are numărul `1`). La un moment dat, o singură persoană poate intra în cameră, poate servi un pahar de `1` dl de Pepsi, din una dintre sticlele existente în cameră, îl bea, apoi pleacă. Există două categorii de persoane: persoane risipitoare (care întotdeauna vor lua un pahar din una dintre cele mai pline sticle) și persoane econoame (care întotdeauna vor lua un pahar din una dintre cele mai goale sticle; evident, sticlele complet goale se ignoră).

Cunoscând volumul de lichid rămas în fiecare dintre cele `M` sticle, scrieți un program care să determine sticla din care a băut fiecare persoană.

Date de intrare/ieșire

Fișierul de intrare `pepsi.in` conține pe prima linie trei numere naturale `N M K`, separate prin spații, reprezentând numărul de persoane așezate la coadă, numărul

de sticle și, respectiv, numărul de decilitri de suc existenți inițial în fiecare dintre cele M sticle (1≤N≤1000, 1≤M≤100, 1≤K≤20). Cea de-a doua linie conține o succesiune de N caractere ce pot fi R sau E; dacă al i-lea caracter din această succesiune este R, deducem că persoana i este risipitoare, altfel, că este econoamă. Cea de-a treia linie din fișierul de intrare conține M numere naturale separate prin spații, reprezentând, în ordine, volumul de lichid rămas în fiecare dintre cele M sticle.

Fișierul de ieșire pepsi.out conține o singură linie pe care se află N numere naturale separate prin spații; al i-lea număr reprezintă numărul sticlei din care a băut persoana i. Dacă nu există soluție, pe prima linie a fișierului de ieșire se va scrie mesajul IMPOSIBIL.

Exemple

pepsi.in	pepsi.out	pepsi.in	pepsi.out
4 3 3 EEEE 3 0 2	2 2 2 3	5 5 2 EERRE 1 2 0 2 0	3 3 5 1 5

26. *Bonus*

Gigel este programator amator, dar web designer profesionist. Acum proiectează un magazin virtual și se confruntă cu o problemă de programare. Pe site sunt prezentate n produse, numerotate de la 1 la n. În coșul de cumpărături virtual, un client poate alege orice submulțime de produse dintre cele n. În funcție de componența coșului de cumpărături, Gigel trebuie să ofere un bonus.

Gigel a observat că se pot forma din cele n produse 2^n-1 submulțimi distincte nevide. A ordonat submulțimile lexicografic și a asociat apoi fiecărei submulțimi un număr de ordine de la 1 la 2^n-1.

De exemplu, pentru n=3 submulțimile nevide în ordine lexicografică sunt:

Submulțimea	{1}	{1, 2}	{1, 2, 3}	{1, 3}	{2}	{2, 3}	{3}
Nr. de ordine	1	2	3	4	5	6	7

Pentru a acorda corect și eficient bonusul, trebuie să fie implementate operațiile:
- dată fiind o submulțime, să determine numărul ei de ordine;
- dat fiind un număr de ordine, să determine submulțimea corespunzătoare.

Date de intrare/ieșire

Fișierul de intrare bonus.in conține pe prima linie numărul natural n (1≤n≤30), iar pe a doua linie, un număr natural m, reprezentând numărul de operații ce trebuie executate (1≤m≤1000). Pentru fiecare operație urmează în fișier câte două linii; pe prima linie din cele două este scris tipul operației (1 sau 2); dacă tipul operației este 1, atunci pe cea de-a doua linie este scris numărul de elemente din submulțime, urmat de elementele submulțimii, separate prin spații; dacă tipul operației este 2, pe cea de-a doua linie va fi scris un număr de ordine (adică un număr natural cuprins între 1 și 2^n-1).

Fişierul de ieşire `bonus.out` va conține `m` linii, câte una pentru fiecare operație din fişierul de intrare. Pe fiecare linie va fi scris rezultatul unei operații, în ordinea în care operațiile apar în fişierul de intrare. Dacă operația este de tip `1`, în fişierul de ieşire va fi afişat numărul de ordine al submulțimii specificate în operație. Dacă operația este de tip `2`, în fişierul de ieşire vor fi afişate elementele submulțimii cu numărul de ordine specificat, separate prin spații, în ordine crescătoare.

Exemple

bonus.in	bonus.out
3 3 2 3 1 2 2 3 2 7	1 2 3 6 3

(Olimpiada Municipală de Informatică, Iaşi, 2004)

27. *Sumă Fibonacci*

Considerăm şirul lui Fibonacci: `0, 1, 1, 2, 3, 5, 8, 13, ...` Dat fiind un număr natural cu maxim 80 de cifre, scrieți acest număr sub formă de sumă de termeni distincți neconsecutivi din şirul Fibonacci, astfel încât numărul de termeni din sumă să fie minim. De exemplu, pentru `n=20` veți afişa `2+5+13`.

(Olimpiada Națională de Informatică, Constanța, 2000)

28. *Pic*

Un grup de cercetători pleacă într-o expediție în Oceanul Pacific. După o furtună puternică rămân fără apă. Norocul vine pe neaşteptate printr-o ploaie torențială. Toate cele `n` vase existente pe vapor sunt scoase şi umplute cu apă (`n<70`). Fiecare vas în care se reține apa de ploaie dispune de un dispozitiv care contorizează picăturile de apă, prin urmare, pentru fiecare vas `i` se cunoaşte numărul de picături din vas x_i ($x_i<10^{22}$). După ce se termină ploaia, apa strânsă este mutată în cele `k` butoaie (toate goale) existente pe vapor (`k<40`). Se cunoaşte capacitatea fiecărui butoi – în butoiul `i` încap v_i ($v_i<10^{22}$) picături, $1 \le i \le k$. Să se determine numărul maxim de butoaie care se pot umple.

(*.campion*, 2003)

29. *Maşină*

Vasile a „împrumutat" o maşină şi a plecat în oraş să se distreze. Ghinionul a fost că maşina aparținea poliției şi era dotată cu un echipament care emitea semnale indicând direcția de mişcare a maşinii. Polițiştii au luat imediat harta oraşului şi, cunoscând poziția inițială a maşinii şi direcțiile de mişcare, vor să determine poziția finală a maşinii. Harta oraşului este reprezentată ca o matrice cu `R` linii şi `C` coloane, în matrice fiind marcate cu `'.'` (punct) pozițiile prin care maşinile pot

circula, iar cu 'X' pozițiile prin care mașinile nu pot circula. Poziția inițială a mașinii a fost marcată pe hartă cu '*'. Mașina se poate deplasa în patru direcții: la nord (adică de pe linia i pe linia i-1 pe hartă), la sud (adică de pe linia i pe linia i+1 pe hartă), la vest (adică de pe coloana i pe coloana i-1 pe hartă) sau la est (adică de pe coloana i pe coloana i+1 pe hartă).

Scrieți un program care să ajute polițiștii să găsească mașina.

Date de intrare/ieșire

Pe prima linie a fișierului de intrare masina.in se găsesc două numere naturale separate prin spațiu R C (1≤R, C≤50), unde R reprezintă numărul de linii, iar C numărul de coloane de pe harta orașului. Fiecare dintre următoarele R linii conține o secvență de C caractere care pot fi doar '.', '*' sau 'X', descriind harta orașului. Următoarea linie, a (R+2)-a, conține un număr natural N (1≤N≤1000) reprezentând numărul de schimbări de direcție pe care le-a făcut mașina. Fiecare dintre următoarele N linii conține un șir de caractere ce poate fi NORD, SUD, VEST sau EST. Oricare două direcții consecutive sunt diferite.

Fișierul de ieșire masina.out conține harta orașului reprezentată ca în fișierul de intrare (R linii de câte C caractere). Pe hartă vor fi marcate cu '*' numai pozițiile finale posibile ale mașinii.

Exemple

masina.in
```
3 4
....
*..X
X.X.
2
EST
NORD
```

masina.out
```
.**.
...X
X.X.
```

masina.in
```
10 9
........X
X..XX..X.
.X.XX.X..
...XX....
...XX....
.XXX..XX.
.......X.
..XXX.X..
X.X....X.
*.....X..
4
EST
NORD
EST
SUD
```

masina.out
```
........X
X..XX.*X.
.X.XX.X..
...XX....
...XX.***
.XXX..XX*
.......X*
..XXX*X.*
X.X..*.X*
....**X.*
```

Anexe

1. Tabela codurilor ASCII

Cod	Caracter	Cod	Caracter	Cod	Caracter	Cod	Caracter	Cod	Caracter	Cod	Caracter
000	(nul)	022	■ (syn)	044	,	066	**B**	088	**X**	110	**n**
001	☺ (soh)	023	¥ (etb)	045	-	067	**C**	089	**Y**	111	**o**
002	☻ (stx)	024	↑ (can)	046	.	068	**D**	090	**Z**	112	**p**
003	♥ (etx)	025	↓ (em)	047	/	069	**E**	091	**[**	113	**q**
004	♦ (eot)	026	→ (eof)	048	**0**	070	**F**	092	\	114	**r**
005	♣ (enq)	027	← (esc)	049	**1**	071	**G**	093	**]**	115	**s**
006	♠ (ack)	028	⌐ (fs)	050	**2**	072	**H**	094	^	116	**t**
007	• (bel)	029	↔ (gs)	051	**3**	073	**I**	095	_	117	**u**
008	_ (bs)	030	▲ (rs)	052	**4**	074	**J**	096	`	118	**v**
009	□ (tab)	031	▼ (us)	053	**5**	075	**K**	097	**a**	119	**w**
010	◙ (lf)	032	(spațiu)	054	**6**	076	**L**	098	**b**	120	**x**
011	♂ (vt)	033	!	055	7	077	**M**	099	**c**	121	**y**
012	♀ (np)	034	"	056	**8**	078	**N**	100	**d**	122	**z**
013	♪ (cr)	035	#	057	**9**	079	**O**	101	**e**	123	{
014	♫ (so)	036	$	058	:	080	**P**	102	**f**	124	|
015	☼ (si)	037	%	059	;	081	**Q**	103	**g**	125	}
016	► (dle)	038	&	060	<	082	**R**	104	**h**	126	~
017	◄ (dc1)	039	'	061	=	083	**S**	105	**i**	127	
018	↕ (dc2)	040	(	062	>	084	**T**	106	**j**		
019	‼ (dc3)	041	)	063	?	085	**U**	107	**k**		
020	¶ (dc4)	042	*	064	@	086	**V**	108	**l**		
021	§ (nak)	043	+	065	**A**	087	**W**	109	**m**		

Codul **ASCII** extins

Cod	Caracter	Cod	Caracter	Cod	Caracter	Cod	Caracter	Cod	Caracter	Cod	Caracter
128	Ç	149	ò	170	¬	191	┐	212	╘	234	Ω
129	ü	150	û	171	½	192	└	213	╒	235	δ
130	é	151	ù	172	¼	193	┴	214	╓	236	∞
131	â	152	_	173	¡	194	┬	215	╫	237	∅
132	ä	153	Ö	174	«	195	├	216	╪	238	∈
133	à	154	Ü	175	»	196	─	217	┘	239	∩
134	å	155	¢	176	▒	197	┼	218		240	≡
135	ç	156	£	177	▒	198	╞	219	▉	241	±
136	ê	157	¥	178	▓	199	╟	220	▄	242	≥
137	ë	158	_	179	│	200	╚	221	▌	243	≤
138	è	159	č	180	┤	201	╔	222	▐	244	⌠
139	ï	160	á	181	╡	202	╩	223	▀	245	⌡
140	î	161	í	182	╢	203	╦	224	α	246	÷
141	ì	162	ó	183	╖	204	╠	225	ß	247	≈
142	Ä	163	ú	184	╕	205	═	226	Γ	248	°
143	Å	164	ñ	185	╣	206	╬	227	π	249	•
144	É	165	Ñ	186	║	207	╧	228	Σ	250	–
145	æ	166	ª	187	╗	208	╨	229	σ	251	√
146	Æ	167	º	188	╝	209	╤	230	μ	252	ⁿ
147	ô	168	¿	189	╜	210	╥	231	τ	253	²
148	ö	169	_	190	╛	211	╙	232	ϕ	254	■
								233	Θ	255	

2. Cuvintele cheie C/C++

asm	_asm	__asm	auto	break	case
cdecl	_cdecl	__cdecl	char	class	const
continue	_cs	__cs	default	delete	do
double	_ds	__ds	else	enum	_es
__es	_export	__export	extern	far	_far
__far	_fastcall	__fastcall	float	for	friend
goto	huge	_huge	__huge	if	inline
int	interrupt	_interrupt	__interrupt	_loadds	__loadds
long	near	_near	__near	new	operator
pascal	_pascal	__pascal	private	protected	public
register	return	_saveregs	__saveregs	_seg	__seg
short	signed	sizeof	_ss	__ss	static
struct	switch	template	this	typedef	union
unsigned	virtual	void	volatile	while	

3. Sisteme de numerație

Un *sistem de numerație* este alcătuit dintr-o mulțime finită de simboluri și un set de reguli de reprezentare a numerelor cu ajutorul simbolurilor respective. Numărul de simboluri constituie *baza* sistemului de numerație.

În general, utilizăm sisteme de numerație poziționale, pentru care pozițiile simbolurilor corespund puterilor bazei sistemului de numerație. Mulțimea simbolurilor utilizate pentru reprezentarea numerelor depinde de baza sistemului de numerație. De exemplu, în sistemul de numerație *zecimal* se utilizează zece simboluri: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; în sistemul de numerație *binar* se utilizează două simboluri: 0, 1; în sistemul de numerație *octal* se utilizează opt simboluri: 0, 1, 2, 3, 4, 5, 6, 7; în sistemul de numerație *hexazecimal* se utilizează 16 simboluri: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Semnificația zecimală a simbolurilor A, B, C, D, E, F este, în ordine, 10, 11, 12, 13, 14, 15.

Să notăm cu b baza sistemului de numerație ($b \in N, b>1$).

Regula de reprezentare a numerelor în baza b

Fiecare cifră dintr-un număr valorează de b ori mai mult decât cifra din dreapta sa, deci cifrele numărului, de la dreapta spre stânga, corespund pozițional puterilor bazei sistemului de numerație:

$$x_n x_{n-1} \ldots x_1 x_0 , y_1 y_2 \ldots y_p{}_{(b)} = x_n \cdot b^n + x_{n-1} \cdot b^{n-1} + \ldots + x_1 \cdot b + x_0 + y_1 \cdot b^{-1} + \ldots + y_p \cdot b^{-p}$$

Operații de conversie

Conversia numerelor din baza 10 în baza b

Regula de conversie a numerelor naturale

Pentru a converti un număr natural din baza 10 într-o bază b ($b \in N, b>1$) vom face împărțiri succesive la b până când obținem câtul 0. La primul pas, deîmpărțitul este numărul care se convertește; în continuare, la fiecare pas, câtul de la împărțirea precedentă devine deîmpărțit. Reprezentarea numărului în baza b se obține considerând resturile în ordinea inversă obținerii lor.

Exemplu

Pentru a converti $n=1263_{(10)}$ în baza 16 vom face împărțiri succesive la 16:

deîmpărțit		cât		rest
1263	=16*	78	+	15
78	=16*	4	+	14
4	=16*	0	+	4

Considerând resturile în ordine inversă, obținem $1263_{(10)}=4EF_{(16)}$.

Regula de conversie a numerelor reale din intervalul (0,1)

Conversia numerelor subunitare din baza 10 într-o bază b ($b \in N, b>1$) se realizează prin înmulțiri succesive cu b, până când se obține o perioadă sau până când partea fracționară a rezultatului este 0. La primul pas, deînmulțitul este numărul care se convertește; în continuare, la fiecare pas, partea fracționară a rezultatului precedent devine deînmulțit. Reprezentarea numărului în baza b se obține considerând cifrele de la partea întreagă a rezultatelor în ordinea obținerii lor.

Exemplu

Să convertim $n=0,3125_{(10)}$ în baza 2:

```
0,3125  x 2 = 0,625
0,625   x 2 = 1,25
0,25    x 2 = 0,5
0,5     x 2 = 1,0
```

Am obținut partea fracționară 0, deci $0,3125_{(10)}=0,0101_{(2)}$.

Regula de conversie a numerelor reale

Pentru a converti un număr real din baza 10 în baza b ($b\in N$, $b>1$) se convertesc separat partea întreagă și partea fracționară, conform regulilor anterioare.

De exemplu, $1263,3125_{(10)}=10011101111,0101_{(2)}$.

Conversia numerelor din baza b în baza 10

Pentru a converti un număr din baza b ($b\in N$, $b>1$) în baza 10 utilizăm dezvoltarea după puterile bazei sistemului de numerație și efectuăm calculele.

Exemple

$1AF_{(16)}=1\cdot16^2+A\cdot16+F=431_{(10)}$

$11001,101_{(2)}=2^4+2^3+1+2^{-1}+2^{-3}=25,625_{(10)}$

Conversia numerelor din baza b în baza b'

Conversia numerelor din baza b în baza b' ($b, b'\in N$, $b, b'>1$) se realizează prin intermediul bazei 10: realizăm conversia din baza b în baza 10, apoi din baza 10 în b'.

Conversii între baze puteri ale lui 2

Datorită faptului că, deocamdată, calculatoarele utilizează informații reprezentate în formă binară, sistemul de numerație binar prezintă pentru informaticieni o importanță deosebită. Calculatoarele par a se descurca foarte ușor cu șiruri de cifre binare, dar oamenii se obișnuiesc mai greu. Frecvent, informaticienii apelează la conversia șirurilor binare în valori octale sau hexazecimale[40]. Operațiile de conversie între bazele 2 și 8, respectiv 2 și 16 se realizează mai simplu decât conversiile între două baze oarecare, datorită faptului că atât 8, cât și 16 sunt puteri ale bazei 2.

Regula de conversie a numerelor din baza 8 (16) în baza 2

Pentru a converti un număr octal (hexazecimal) în binar vom înlocui fiecare cifră a numărului cu reprezentarea sa binară pe 3, respectiv 4 poziții.

Exemple

$168AF_{(16)} = 1\ 0110\ 1000\ 1010\ 1111_{(2)}$

$75210,231_{(8)} = 111\ 101\ 010\ 001\ 000,\ 010\ 011011_{(2)}$

40. De exemplu, informaticienii exprimă uzual adresele de memorie în hexazecimal, nu ca un șir de 32 de cifre binare.

Regula de conversie a numerelor din baza 2 în baza 8 (16)

Pentru a converti un număr binar în baza 8 (16) vom asocia cifrele binare în grupe de câte 3 (respectiv 4) începând de la dreapta spre stânga pentru partea întreagă şi de la stânga la dreapta pentru partea zecimală, completând ultima grupă cu zerouri nesemnificative.

Exemple

1. 110 1010 1110 1010 1000 0010, 1111 0000 0001 $11_{(2)}$=6AE82,F01C$_{(16)}$
2. 1 111 001 010 000 110, 110 101 $1_{(2)}$=171206,654$_{(8)}$

Pentru aceste conversii este util tabelul următor, care conţine reprezentarea binară a valorilor {0, 1, ..., 15}:

Sistem zecimal	Sistem binar	Sistem zecimal	Sistem binar	Sistem zecimal	Sistem binar	Sistem zecimal	Sistem binar
0	0	4	100	8	1000	12	1100
1	1	5	101	9	1001	13	1101
2	10	6	110	10	1010	14	1110
3	11	7	111	11	1011	15	1111

4. Organizarea logică a memoriei

Din punct de vedere logic, memoria poate fi privită ca o succesiune de *celule binare*, fiecare celulă fiind capabilă să reţină o cifră binară (*0* sau *1*).

Cantitatea de informaţie ce poate fi înregistrată într-o celulă binară se numeşte *bit* (*BI*nary digi*T* – cifră binară).

Principalele operaţii de lucru cu memoria sunt extragerea informaţiilor din memorie (*citire*) şi transferul informaţiilor în memorie (*scriere*). Pentru realizarea acestor operaţii este necesară localizarea zonei de memorie ce conţine informaţiile (pentru citire), respectiv a zonei de memorie în care urmează a fi stocate informaţiile (pentru scriere). Localizarea unei zone de memorie se realizează prin intermediul unei *adrese*.

Cea mai mică zonă de memorie adresabilă este *locaţia de memorie*, constituită din 8 celule binare consecutive[41].

Cantitatea de informaţie stocată într-o succesiune de 8 celule binare se numeşte *octet* (*byte*).

Numărul total de *bytes* care pot fi înregistraţi în memorie reprezintă *capacitatea memoriei*.
Pentru a exprima capacitatea memoriei se folosesc multipli ai *byte*-ului:

1 Kb (kilobyte) = 1024 bytes (2^{10} bytes)
1 Mb (megabyte) = 1024 Kb (2^{20} bytes)
1 Gb (gigabyte) = 1024 Mb (2^{30} bytes)
1 Tb (terabyte) = 1024 Gb (2^{40} bytes).

41. Uzual, termenul *byte* desemnează atât locaţia de memorie, cât şi informaţia conţinută.

5. Reprezentarea datelor în memorie

Oamenii comunică prin intermediul unor simboluri. Din aceste simboluri (de exemplu, literele alfabetului), respectând anumite reguli sintactice, se pot obține construcții mai ample (de exemplu, cuvinte, propoziții, fraze), purtătoare de informație. Dar mintea umană este mult mai complexă decât un calculator. Calculatorul nu este capabil să utilizeze decât două simboluri (0 și 1). Pentru a reuși să comunice cu un calculator, omul trebuie să convertească datele și comenzile sale într-o formă pe care calculatorul să o poată înțelege (formă binară). O astfel de operație se numește *codificare*.

Codificarea caracterelor

Codificarea caracterelor constă în a asocia în mod unic fiecărui caracter (litere, cifre, semne speciale) o valoare numerică.

Cel mai utilizat cod alfanumeric este codul *ASCII* (*A*merican *S*tandard *C*ode for *I*nformation *I*nterchange), prin care se asociază caracterelor un cod format din 7 cifre binare, deci o valoare numerică din mulțimea {0, 1, 2, ..., 127}. De exemplu, codul caracterului 'A' este 65. Pentru reprezentarea unor simboluri suplimentare (caractere specifice altor limbi, simboluri matematice, simboluri grafice), se utilizează codul *ASCII extins*. Codul ASCII extins utilizează 8 biți pentru codificarea caracterelor, permițând astfel codificarea a 256 de simboluri distincte.

Codificarea datelor numerice

Modul de codificare a datelor numerice și dimensiunea zonei de memorie necesare depind de tipul acestora.

1. Codificarea numerelor naturale

Numerele naturale sunt codificate prin reprezentarea lor binară. Dimensiunea reprezentării este fixă, fiind utilizate 8, 16, 32 sau 64 de poziții binare, în funcție de domeniul de valori.

2. Codificarea numerelor întregi

Numerele întregi se reprezintă în *cod complementar*. Dimensiunea reprezentării poate fi de 8, 16, 32 sau 64 de poziții binare, în funcție de domeniul de valori de reprezentat:

Numerele întregi pozitive sunt deci codificate prin reprezentarea lor binară.

Numerele întregi negative se codifică prin complementarierea codificării valorii lor absolute față de 2^x, unde x reprezintă dimensiunea reprezentării.

Observați că prima poziție din reprezentare precizează semnul (primul bit este 0 pentru numere pozitive, respectiv 1 pentru numere negative).

Exemplu

Pentru a reprezenta în cod complementar numărul întreg `-125`, se reprezintă în binar valoarea sa absolută (`125`), apoi se calculează complementul față de 2^8 (prin scădere binară din 2^8), 8 poziții binare fiind suficiente pentru a reprezenta numărul `-125`.

```
1 0 0 0 0 0 0 0 0-
  0 1 1 1 1 1 0 1
  1 0 0 0 0 0 1 1
```

3. Codificarea numerelor raționale

Numerele raționale se codifică uzual folosind reprezentarea în *virgulă mobilă*, utilizând 32 de biți, 64 de biți sau 80 de biți. Pentru a reprezenta un număr rațional în virgulă mobilă se convertește numărul în binar, apoi se aduce în formă normalizată. Pentru a aduce numărul în formă normalizată se înmulțește numărul cu o putere convenabilă a lui 2 (baza sistemului de numerație), astfel încât partea întreagă a numărului să fie 1. Deci, ca urmare a operației de normalizare, obținem: $N = \pm 1, a_1a_2...a_n \cdot 2^{\pm e}$.

Exemplu

Să considerăm numărul rațional $10,625_{(10)}$. Convertind numărul în baza 2, obținem: $1010,101$. Pentru a aduce numărul în formă normalizată trebuie să deplasăm trei cifre de la partea întreagă la partea fracționară, ceea ce echivalează cu o împărțire cu 2^3. Deci: $10,625_{(10)}=1010,101_{(2)}= 1,010101_{(2)} \cdot 2^3$.

Reprezentarea numerelor raționale în *virgulă mobilă simplă precizie* (32 de biți) are următoarea structură:

- bitul cel mai semnificativ precizează semnul numărului rațional;
- următorii 8 biți reprezintă caracteristica;
- următorii 23 de biți reprezintă partea fracționară a numărului rațional în formă normalizată.

Deoarece exponentul bazei de numerație în forma normalizată poate fi și negativ, pentru a nu mai utiliza o poziție specială pentru semnul exponentului, în reprezentarea în virgulă mobilă se utilizează o mărime denumită *caracteristică* $C=e+127$. Dacă $C \in \{0, 1, ..., 126\}$, exponentul e este negativ; dacă $C \in \{127, ..., 255\}$, exponentul este pozitiv. Dacă partea fracționară a numărului rațional nu necesită 23 de poziții binare, alinierea se face la stânga, completându-se, eventual, restul zonei de memorie cu zerouri nesemnificative.

Probleme speciale apar la reprezentarea numărului rațional `0.0`, care nu poate fi adus în formă normalizată. În acest caz, caracteristica este considerată `0`, iar partea fracționară de asemenea `0`, semnul fiind + sau −. Deci `0` admite două reprezentări (+0 și −0). Din acest motiv, valoarea `-127` nu face parte din domeniul de valori valide ale exponentului, deși ar putea fi reprezentată.

6. Etapele dezvoltării programelor

Dezvoltarea unei aplicații comportă mai multe etape:

1. *Editarea textului programului.* Pentru această operație este necesar un *editor de texte.* Rezultatul este un fișier sursă, care are extensia `.cpp` (pentru C++) sau `.c` (pentru C).
2. *Compilarea programului.* Pentru această operație se lansează în execuție un program special, denumit *compilator.* Compilatorul analizează textul sursă al programului din punct de vedere sintactic, semnalând eventualele erori. Dacă programul nu conține erori sintactice, compilatorul traduce acest program din limbajul de programare utilizat de programator în singurul limbaj „înțeles" de calculator – limbajul-mașină. Rezultatul compilării este un fișier denumit fișier obiect, care are extensia `.obj` (sub Windows/DOS) sau `.o` (sub Linux).

3. *Editarea de legături.* Această operație este realizată de un program special denumit *editor de legături* (sau *link-editor*). Editorul de legături asamblează toate modulele obiect din care este constituită aplicația, rezultând programul executabil (sub Windows/DOS, un fișier cu extensia `.exe`).

Soluții și indicații

1. Elemente de bază ale limbajului C/C++

1. `a=13; b=7`. **2.** `c=3412`. **3.** `-78`. **4.** d. **5. a.** La declarare, nu se pot inițializa simultan două variabile. **b.** Variabila `x` nu este definită. **d.** La declarare, variabilele se separă prin virgulă. **f.** Nu există tipul `unsigned float`. **6.** c. **7.** e. **8. a.** `2.5`; **b.** `32767`; **c.** `-1`; **d.** `8`; **e.** `1`; **f.** `da`; **g.** `16`; **h.** `61`. **9. a.** Operatorul `%` nu se aplică operanzilor reali. **b.** Ambiguitate: se adună `a` cu valoarea incrementată a lui `x` sau se adună `a` cu `x` și apoi se incrementează valoarea lui `a`? **c.** Între operanzii `4`, `a` și `c` (sau `ac`) lipsesc operatorii, deci `4ac` este interpretat ca nume incorect de variabilă. **d.** Operatorii logici pe biți nu se aplică operanzilor reali. **e.** Variabila `c` este de tip `char`, nu i se poate atribui un șir de caractere. **f.** Nu există operatorul `&&=`; **10.** f. **11.** c. **12.** a și b. **13.** c. **14.** d. **15.** d. **16.** `c=(c>='a'&&c<='z'? c-'a'+'A':c)` **17. a.** `x<<=n;` **b.** `x>>=n;` **c.** `x&1;` **d.** `x&1<<n;` **e.** `x&=~(1<<n);` **f.** `x|=1<<n`.

18.

```
#include <iostream.h>
#include <math.h>
int main()
{ float l; cout << "Latura="; cin >> l;
  cout << "Inaltimea=" << l*sqrt(3)/2 << endl;
  cout << "Aria=" << l*l*sqrt(3)/4 << endl; return 0;}
```

19. `aux=x1; x1=x2; x2=x3; x3=x4; x4=x5; x5=aux;`

20.

```
/* transform distanta in metri si timpul in secunde */
D=D*1000; H=H*3600;
v=D/H;
```

21. Utilizăm formula vitezei `v=s/t` ⇒ `t=s/v`. Cei doi colegi se întâlnesc după același interval de timp, în care primul a parcurs distanța `d1`, iar cel de-al doilea distanța `D-d1`. Obținem un sistem de ecuații prin rezolvarea căruia determinăm `d1` și `t`:

```
d1=v1*D/(v1+v2); t=d1/v1;
```

22. `L=sqrt((xA-xB)*(xA-xB)+(yA-yB)*(yA-yB));`

23. `cout<<(x<=10?2>x*x?sqrt(2-x*x):sqrt(x*x-2):(-2*x+1)/3);`

2. Instrucțiunile limbajului C/C++

2. Valoarea afișată este `3`. **3.** `5`.

4.

```
int a, b;
cin>>a>>b;
if (a-b==1) || (b-a==1) cout<<"DA";
   else cout<<"NU";
```

5.
```
float a, b, c, s, p, d;
s=-b/a; p=c/a; d=b*b-4*a*c;
if (d<0) cout<<"Ecuatia nu are radacini reale"<<endl;
   else
    {cout"Ecuatia are radacini reale"<<endl;
     if (d==0) cout<<"Radacinile sunt egale"<<endl;
       else cout<<"Radacinile sunt distincte"<<endl;
     if (p<0) cout<<"Radacinile au semne contrare"<<endl;
       else
         {cout<<"Radacinile sunt "<<endl;
          if (s>0) cout<<"pozitive"<<endl;
            else
              if (s<0) cout<<"negative"<<endl;
                else cout<<"nule"<<endl;
         }
    }
```
6.
```
unsigned x, c1, c2, c3;
cin>>x;
c3←x%10; c2←x/10%10; c1←x/100; /* extrag cele 3 cifre ale numarului */
if (c1<c2) cout<<c2<<c3;
   else if (c2<c3) cout<<c1<<c3;
           else cout<<c1<<c2;
```
7.
```
char c1, c2, c3;
cin>>c1>>c2>>c3;
if (c1==c2) && (c2==c3) nr=1;
   else if (c1==c3) || (c1==c2) || (c2==c3) nr=2;
          else nr=3;
 cout<<nr;
```
8.
```
float ag, am, as, bg, bm, bs, cg, cm, cs;
cin>>ag>>am>>as>>bg>>bm>>bs;
cs=as+bs; cm=0; cg=0;
if (cs>60) {cs=cs-60; cm=1;}
cm=cm+am+bm;
if (cm>60) {cm=cm-60; cg=1;}
cg=cg+ag+bg;
if (cg>360) cg=cg-360;
cout<<cg<< "grade "<<cm<<" minute "<<cs<<" secunde";
```
10.
```
{float x, y;
 cin>>x>>y;
 if (x>0 && y>0) cout<<(x+y)/(2*x*y);
   else
     if (x==0 || y==0)
        if (x>y) cout<<x;
            else cout<<y;
       else cout<<(1/x+1/y)*(1/x-1/y+2*x*y+x*x+y*y);}
```
11. Conform restricțiilor problemei există doar 3 variante în care pot fi vizitate cele 3 orașe (ABC, ACB, BAC). Vom calcula lungimea fiecărui traseu posibil și vom afișa traseul cel mai scurt.
```
{unsigned a, b, c;
 cin>>a>>b>>c;
 if (a+b<c+b && a+b<a+c) cout<<"ABC";
   else if (b+c<a+b && b+c<a+c) cout<<"ACB";
          else cout<<"BAC";}
```
Observați că nu am utilizat nici o variabilă suplimentară.

12.
```
float a1, b1, c1, a2, b2, c2, x, y;
cin>>a1>>b1>>c1>>a2>>b2>>c2;
if (a1==0)
   if (b1==0)
     if (c1==0)
       cout<<"x si y pot avea orice valoare reala";
     else cout<<"Sistemul nu are solutii";
   else      /* a1=0, b1≠0 */
     if (a2==0)
       if (c1*b2==c2*b1)
         cout<<"y="<<c1/b1<<" x orice valoare reala";
       else cout<<"Sistemul nu are solutii";
     else /* a2≠0 */
       cout<<"x="<<(c2*b1-c1*b2)/(a2*b1)<<" y="<<c1/b1;
 else      /* a1≠0 */
   if (a1*b2==a2*b1)
     if (a1*c2==c1*a2)
       cout<<"x si y pot avea orice valoare reala";
     else
       cout<<"Sistemul nu are solutii";
   else
     {y=(c2*a1-a2*c1)/(b2*a1-a2*b1);
      x=(c1-b1*y)/a1;
      cout<<"x= "<<x<<" y="<<y;}
```

13.
```
{float m1, m2, m3, m4, m5, mg;
 cin>>m1>>m2>>m3>>m4>>m5;
 if (m1<5 || m2<5 || m3<5 || m4<5 || m5<5)
   cout<<"Nepromovat";
   else
     {mg=(m1+m2+m3+m4+m5)/5;
      if (mg<6) cout<<"Nepromovat";
        else cout<<"Promovat";}
}
```

14.
```
{unsigned h1, h2, h3;
 cin>>h1>>h2>>h3;
 if (h1≤h2)
   if (h2≤h3) cout<<"Ionel Gigel Danut"
   else  /* h3<h2 */
     if (h1≤h3) cout<<"Ionel Danut Gigel"
     else /* h3<h1 */
       cout<<"Danut Ionel Gigel"
 else /* h2<h1 */
   if (h1≤h3) cout<<"Gigel Ionel Danut";
   else /*h2<h1, h3<h1 */
     if (h2≤h3) cout<<"Gigel Danut Ionel"
     else /* h3<h2<h1*/
       cout<<"Danut Gigel Ionel";}
```

15. **a.** 3; **b.** 4321; **c.** Valoarea lui n este scăzută succesiv cu 1 şi afişată, cât timp n≠0. Prin scăderi succesive cu 1 se va ajunge la situaţia în care se va scădea 1 din -32768, obţinându-se +32767, care se va scădea în continuare cu 1 până la 0. Deci, se vor afişa pe ecran 65535 de numere; **d.** 051015; **e.** 01 02 03 04 12 13 14 23 24 34; **f.** ciclul este infinit, afişându-se pe ecran numai numere impare ; **g.** -4; **h.** afişează primele n numere Fibonacci: 1 1 2 3 5 8 13 21. **16.** a şi c. **17.** e şi f. **18.** a, d şi f. **19.** 30. **20.** b şi c. **21.** a, b şi d. **22.** c. **23.** d. **24.** c. **25.** d. **26.** **a.** 4368 **b.** algoritmul extrage succesiv cifrele din numerele a şi b, până când unul dintre numere se termină; la fiecare pas compară ultima cifră din a cu ultima cifră din b, o alege pe cea mai mare şi o adaugă la sfârşitul numărului

construit în variabila c, prin urmare o soluție posibilă ar fi 8291; **c.** o soluție posibilă ar fi a=10, b=10 (sau orice putere a lui 10); **27. a.** 5; **b.** orice număr pentru care cel mai mare divizor prim este 11, de exemplu 11 sau 33; **c.** calculează cel mai mare divizor prim al lui n. **28. a.** 78; **b.** 9; **c.** numărul n, considerat a fi scris în baza b, este convertit în baza 10. **29. a.** 55; **b.** a=4 și b=7; **c.** orice pereche de valori, pentru care a>b; **d.** în variabila x se calculează suma numerelor din intervalul [a,b]; calculul acestei valori se poate realiza direct, astfel:

```
if (a<=b) x=b*(b+1)/2-a*(a-1)/2;
  else x=0;
```

30. d. **31.** c și d. **32.** 5. **33.** Variabila m nu a fost inițializată cu 0. Ciclul este infinit deoarece se execută numai instrucțiunea m++; (dacă am fi dorit să se execute și instrucțiunea n/=p; ar fi trebuit să scriem {m++; n/=p;}, formând astfel o instrucțiune compusă).

35.

```
unsigned a,b,nr,sa,sb,d;
nr=0;      /*numarul de perechi determinate este initial 0*/
a=2;
while (nr<3)              /* nu am obtinut inca 3 perechi */
   {sa=1;                /* calculez suma divizorilor lui a */
    for (d=2; d<=a/2; d++)
      if (a%d==0)               /* d este divizor al lui a */
        sa+=d;                        /* il adunam la suma */
    b=sa;
       /* b trebuie sa fie egal cu suma divizorilor lui a */
    if (b>a)
      {sb=1;              /* calculez suma divizorilor lui b */
       for (d=2; d<=b/2; d++)
         if (b%d==0)                    /*d divizor al lui b*/
           sb=sb+d;                    /* il adunam la suma */
       if (a==sb)/*am obtinut o pereche de numere prietene*/
         {cout<<a<<' '<<b<<endl;                /* o afisez */
          nr++;}                              /* si o numar */
      }
    a++;                                /* caut in continuare */
   }
```

41. Voi citi succesiv valorile date și pentru fiecare valoare citită voi extrage cifrele numărând în variabila nr0 câte cifre egale cu 0 am obținut. **42.** Vom aplica repetat algoritmul de calcul al sumei cifrelor unui număr. **44.** Nu vom parcurge toate numerele naturale mai mici decât n testând la fiecare pas dacă numărul curent este sau nu pătrat perfect, ci vom genera direct pătratele perfecte. **46.** Vom construi toate fracțiile care au numitorul și numărătorul în mulțimea {1,2,...,n} și le vom număra numai pe cele ireductibile. **47.** În mod similar generării celui de-al n-lea termen al șirului Fibonacci, vom utiliza 4 variabile: s0, s1, s2 reține 3 termeni consecutivi din șir, iar în s3 vom calcula termenul următor; după care ne pregătim pentru pasul următor, deplasându-ne în șir s0=s1; s1=s2; s2=s3; **48.** Calculăm mai întâi MO media la oral, apoi media semestrială se obține după formula (3*MO+Teza)/4. **49.** Pentru a calcula c.m.m.d.c. a n valori ne bazăm pe faptul că operația c.m.m.d.c. este asociativă, adică $cmmdc(a_1,\ldots,a_{n-1},a_n)=cmmdc(cmmdc(a_1,\ldots,a_{n-1}),a_n)$. Vom utiliza o variabilă d în care vom memora la fiecare pas c.m.m.d.c. al valorilor citite până la momentul respectiv. Variabila d va fi inițializată cu prima valoare citită. La fiecare pas citim o nouă valoare și calculăm c.m.m.d.c. dintre valoarea citită și d prin algoritmul lui Euclid. **50.** Citim succesiv caracterele în aceeași variabilă c, până la întâlnirea caracterului spațiu. Vom utiliza o variabilă logică (Este) care va avea valoarea adevarat dacă secvența de caractere poate fi un număr natural și fals, altfel. La întâlnirea unui caracter diferit de cifră, variabila Este va primi valoarea fals. Singura modificare pe care trebuie să o facem pentru ca algoritmul să verifice dacă secvența este un număr întreg se referă la primul caracter din secvență (care ar putea fi și '+' sau '-'). Pentru ca numărul să fie real, acceptăm și un singur caracter punct.

51. Citesc propoziția caracter cu caracter, memorând succesiv caracterele citite în aceeași variabilă c:

```
char c;
unsigned nr=0;                              /*numarul de cuvinte*/
do
  {do c=getche(); while (c==' '); /*sar peste spatii */
   if (c!='.')                    /*incepe un cuvant */
     {nr++;                       /*il numar  si sar peste cuvant*/
      while (c!=' ' && c!='.') c=getche();}
  } while (c!='.');
cout<<"nr = "<<nr<<endl;
```

52.

```
{int n, x, r, i;
 cout<<"x, r, n= "; cin>>x>>r>>n;
 for (i=0; i<n; i++) cout<<x+i*r<<' '; }
```

53. Trebuie să calculăm c.m.m.m.c. (1, 2, ..., n); Folosim faptul că c.m.m.m.c.(x,y)=x*y/c.m.m.d.c. (x, y) și asociativitatea c.m.m.m.c.(a_1, a_2, ..., a_n) = c.m.m.m.c.(c.m.m.m.c.(a_1, a_2, ..., a_{n-1}), a_n).

```
int n, i, cmmmc=1, r, x, y;
cout<<"n= "; cin>>n;
for (i=2; i<=n; i++)
     {x=i; y=cmmmc;
      while(x) {r=y%x; y=x; x=r;}
      cmmmc=i*cmmmc/y;}
cout<<cmmmc<<endl;
```

54. Se citesc succesiv cele n numere naturale și se determină pentru fiecare număr citit numărul de divizori primi. Se reține numărul cu număr maxim de divizori primi.

```
{int n, nr_max, nr_div_max=0, snr, d, nr, i, nrd;
 cout<<"n= "; cin>>n;
 for (i=1; i<=n; i++)
   {cout<<"nr="; cin>>nr;              //citesc o noua valoare
    for (snr=nr,d=2,nrd=0;nr>1;d++)    //descompun in factori primi
      if (nr%d==0)                     //am gasit un divizor prim
        {nrd++;                        //il numar
         while (nr%d==0) nr/=d;}
    if (nrd>nr_div_max) nr_max=snr, nr_div_max=nrd; }
cout<<"Numarul maxim de divizori="<<nr_div_max<<endl;
cout<<"Numarul= "<< nr_max;}
```

57. Datorită faptului că n este mare, nu putem calcula produsul celor n numere, pentru ca apoi să numărăm câte cifre 0 are la sfârșit. Dar un 0 final nu se poate obține decât prin înmulțirea lui 2 cu 5, vom calcula pe parcursul citirii multiplicitatea lui 2 (m2) și multiplicitatea lui 5 (m5) în produs. Numărul de zerouri finale este egal cu min(m2, m5). **58.** Fiecare număr x cuprins între 3 și n se testează dacă este sau nu prim. Dacă este prim, se calculează câte cifre are x și se adună la numărul total de cifre pe care trebuie să le scrie bibliotecarul. Dacă x nu este prim, se mărește cu 1 numărul de pagini nemâzgălite.

59. Programul conține trei cicluri for imbricate; contorul primului ciclu reprezintă prima cifră (1... 7), al doilea contor cifra a doua (2... 8), iar ultimul cifra a treia (3... 9).

```
int c1, c2, c3;
for (c1=1; c1<8; c1++)                          //prima cifra
    for (c2=c1+1; c2<9; c2++)                   //cifra a doua
         for (c3=c2+1; c3<10; c3++)             //cifra a treia
              cout <<c1<<c2<<c3<<' ';
```

Numerele „bine ordonate descrescător":

```
for (c1=9; c1>1; c1--)                          //prima cifra
  for (c2=c1-1; c2>0; c2--)                     //cifra a doua
    for (c3=c2-1; c3>=0; c3--)                  //cifra a treia
      cout <<c1<<c2<<c3<<' ';
```

61.

```
int n;
cout << "n= "; cin >> n;
while (n>=1000) {cout <<'M';n-=1000;} //scriu miile
if (n>=900) {cout <<"CM";n-=900;}     //cazul 900
if (n>=500) {cout <<'D';n-=500;}      //cazul 500
if (n>=400) {cout <<"CD";n-=400;}     //cazul 400
while (n>=100) {cout <<'C';n-=100;}   //scriu sutele
if (n>=90) {cout <<"XC";n-=90;}       //cazul 90
if (n>=50) {cout <<'L';n-=50;}        //cazul 50
if (n>=40) {cout <<"XL";n-=40;}       //cazul 40
while (n>=10) {cout <<'X';n-=10;}     //scriu zecile
if (n==9) {cout <<"IX";n-=9;}         //cazul 9
if (n>=5) {cout <<'V';n-=5;}          //cazul 5
if (n==4) {cout <<"IV";n-=4;}         //cazul 4
while (n>0) {cout <<'I';n-=1;}        //scriu unitatile
```

62. Calculăm valoarea celei mai mari bancnote pe care o putem utiliza pentru a plăti suma `S`. Utilizăm numărul maxim de bancnote de valoare maximă, apoi reluăm procedeul pentru restul sumei.

```
int S, x, p=1;
cout << "S= "; cin >> S;  cout << "x= "; cin >> x;
while (p*x<=S) p*=x;          //valoarea celei mai mari bancnote
while (S)                     //mai avem de platit
      {if (S/p)               //pot utiliza bancnote cu valoarea p
          {cout<<S/p<<" bancnote cu valoarea "<<p<<endl;
           S%=p;}             //rest de plata
       p/=x;}                 //trec la bancnote mai mici
```

64. Observăm că dacă un subordonat are un bit cu valoarea 1, orice şef trebuie să aibă bitul corespunzător tot pe valoarea 1. Să notăm cu `nr_1` numărul de biţi 1 din reprezentarea internă a lui `n`. Atunci numărul de zerouri este `nr_0=16-nr_1`. Numărul şefilor lui `n` este egal cu numărul funcţiilor care se pot defini astfel: `f:{1,2,...,nr_0}->{0,1}`. Prin urmare, numărul şefilor este 2^{nr_0}.

```
unsigned n, nr_1=0, i;
cout<<"n= "; cin>>n;
//calculez numarul de biti 1 in reprezentarea lui n
for (i=0; i<16; i++)  nr_1+=(n>>i)&1;
cout <<"Nr. sefilor lui "<<n<<" este "<<(1u<<(16-nr_1));
```

66. Se poate demonstra matematic că pădurarul din poziţia `x0, y0` vede pomul din poziţia `x, y` dacă şi numai dacă `c.m.m.d.c.(|x-x0|, |y-y0|)=1`. Prin urmare, este suficient să numărăm câţi pomi respectă această condiţie>

```
int n, x0, y0, x, y, a, b, r, c=0;
cout << "n, x0, y0= "; cin >> n >> x0 >> y0;
for (x=0; x<n; x++)
    for (y=0; y<n; y++)
        {a = x>x0?x-x0:x0-x; b = y>y0?y-y0:y0-y;
         while (b) {r=a%b;a=b; b=r;}          //calculez cmmdc
         if (a==1) c++;}                      //pom vizibil, il numar
 cout<<"Numarul pomilor vizibili este "<<c<<endl;
```

67. Consider că reprezentarea internă a numerelor mai mici decât `n` (o succesiune de `0` şi `1`) reprezintă un număr în baza `b`, îl convertesc în baza `10` şi îl afişez:

```
int n, i, b, nr=0, x, pb;
 cout<<"n, b ="; cin >> n >> b;
 cout<<0<<' '<<1<<' ';
 for (i=2; ; i++)
   {for (nr=0, x=i, pb=1; x; x>>=1, pb*=b) nr+=(x&1)*pb;
      if (nr<=n) cout<<nr<<' ';  else break;}
```

3. Fişiere

1. Programul afişează calendarul unei luni cu n zile, a fiind ziua din săptămână în care începe luna.

2.
```
ifstream f("nr.in"); if (!fin) return 1;
char c;  int nr=0, lin=1, v;
while (!f.eof())
   {f.get(c);
    if (f.good())
      if (c=='\n')
        {//s-a terminat o linie;
         cout<<"Pe linia "<<lin<<" sunt "<<nr<<" valori\n";
         nr=0; lin++;}
      else
        {f.putback(c); //repun caracterul citit in stream-ul de intrare
         f>>v; nr++;}        //citesc o valoare si o numar
   }
f.close();
```

3. Parcurg secvenţial cele două fişiere, cât timp caracterele citite sunt egale:
```
ifstream f("f1.in"), g("f2.in");
char c1, c2;
while (f.get(c1) && g.get(c2) && c1==c2);
if (!f.get(c1)&& !f.get(c2)) cout << "DA";
      else cout <<"NU";
f.close(); g.close();
```

4.
```
ifstream f("nr.txt"); ofstream g("pare.txt");
int v;
while (!f.eof())
       {f>>v; //citesc o valoare
        if (f.good() && v%2==0)
           g<<v<<endl;} //daca este para o scriu in g
f.close();
g.close();
```

5. Numărul de linii coincide cu numărul de caractere *newline* din fişier.

4. Tipuri structurate de date

1. c, d. **2.** b, d. **3.** d. **4.** e. **5.** b, d. **6.** f. **7.** c. **8.** c. **9.** d. **10.** e. **11. a.** Şirul `s` devine `Era Vysile om frumos`; **b.** Va produce eroare deoarece în membrul drept al unei atribuiri nu poate fi o expresie. **12.** Se va afişa `ErayVayily omxxxumos`. **13.** Se va afişa `bdtcebectdb`. **14.** b, e, f. **15.** d, e, f.

16.
```
double a[10], b[8];
int i, j, nr=0;
for (i=0; i<10; i++) cin >> a[i];                    //citire a
for (i=0; i<8; i++) cin >> b[i];                     //citire b
for (i=0; i<10; i++)                          //parcurg vectorul a
    {//verific daca a[i] este mai mic decat toate elementele din b
      for (j=0; j<8 && a[i]<b[j]; j++);
          if (j==8) nr++;}                            //il numar
cout << nr;
```

O soluţie mai eficientă ar fi să determinăm minimul din vectorul `b`, apoi să comparăm elementele vectorului `a` cu minimul.

18. a.

```
for (ok=1, i=0; ok && i<n; i++)
    {                                          //verific daca a[i] este prim
     for (d=2, rad=sqrt(a[i]); a[i]%d && d<=rad; d++);
         if (a[i]%d==0) ok=0;}                          //a[i] nu este prim
if (ok) cout << "Toate elementele vectorului sunt prime";
   else cout << "Nu toate elementele vectorului sunt prime";
```

b.

```
for (exista=i=0; !exista && i<n; i++)
    {                              //verific daca a[i] este palindrom
      x=a[i]; rx=0;          //determin in rx rasturnatul lui a[i]
      while (x) {rx=rx*10+x%10; x/=10;}
      if (rx==a[i]) exista=1; }              //a[i] este palindrom
if (exista) cout << "Exista un palindrom in vector";
    else  cout << "Nu exista nici un palindrom in vector";
```

c.

```
for (i=0; i<n; )
    if (a[i]%2)//a[i] impar
       {//il elimin
         for (j=i+1; j<n; j++) a[j-1]=a[j];
          n--;}
       else i++;
```

d.

```
for (i=0; i<n; i++)
    if (!a[i])
       {for (j=n; j>i; j--) a[j]=a[j-1]; //fac loc
         n++; //creste numarul de elemente
        a[i+1]=1;}
```

e.

```
int i, lg=1, lgmax=0, emax=a[0], inc=0, incmax=0;
for (i=1; i<n; i++)
    if (a[i]==a[i-1])lg++; //marim lungimea secventei curente
       else
       { if (lg>lgmax) //compar cu secventa de lungime maxima
            {lgmax=lg; incmax=inc; emax=a[i-1];}
         lg=1; inc=i;} //incepe o noua secventa
cout<<lgmax<<' '<<incmax<<' '<<emax;
```

f. Calculez numărul de apariții pentru fiecare element din vector și determin maximul.

19. a.

```
for (exista=j=0; !exista && j<m; j++)
    {//verific daca coloana j are toate elementele nule
     for (i=0; i<n && !a[i][j]; i++);
     if (i==n) exista =1;}                    //coloana j este nula
if (exista) cout <<"Exista o coloana cu toate elementele nule!\n";
   else cout<<"Nu exista o coloana cu toate elementele nule!\n";
```

b.

```
for (i=0;i<n; i++)
    {//calculez produsul elementelor nenule de pe linia i
     for (p=1,j=0; j<m; j++)
         if (a[i][j]) p*=a[i][j];
      cout<<"Produsul elem. nenule de pe linia "<<i<<"="<< p;}
```

c. Citim elementele matricei începând cu linia 1, coloana 1 (lăsam o „bordură" pentru ca orice element din matrice să aibă 4 vecini. Parcurgem apoi matricea și pentru fiecare element testăm cei 4 vecini; dacă toți 4 sunt numere pare, numărăm acest element.

```
#include <fstream.h>
#define NVecini 4
#define DMax 102
int a[DMax][DMax], n, m;
int dl[NVecini]={-1,0,1, 0};
```

```
int dc[NVecini]={ 0,1,0,-1};
int main()
{int i, j, k, nr, total=0;
 ifstream fin("matrice.in");
 fin>>n>>m;
 for (i=1; i<=n; i++) for (j=1; j<=m; j++) fin>>a[i][j];
 fin.close();
 for (i=1; i<=n; i++)
    for (j=1; j<=m; j++)
        {//parcurg vecinii si-i numar pe cei pari
         for (nr=0, k=0; k<NVecini; k++)
            if (a[i+dl[k]][j+dc[k]]%2==0) nr++;
         //daca toti vecinii sunt pari, numar acest element
         if (nr==NVecini) total++;   }
 cout<<total<<endl; return 0;}
```

20. a.

```
//parcurg elementele de deasupra diagonalei principale
for (nule=1,i=0; nule && i<n; i++)
     for (j=i+1; nule && j<n; j++)
          if (a[i][j]) nule=0;
if (nule) cout <<"Matricea este inferior triunghiulara\n";
   else cout <<"Matricea nu este inferior triunghiulara\n";
```

b.

```
//parcurg elementele de sub diagonala principala
for (sim=1, i=0; sim && i<n; i++)
    for (j=0; sim && j<i; j++)
        if (a[i][j]!=a[j][i]) //il compar cu simetricul sau
            sim=0;
if (sim) cout <<"Matricea este simetrica\n";
    else cout <<"Matricea nu este simetrica\n";
```

d. Numărul de chenare concentrice într-o matrice pătratică cu `n` linii este `n/2` pentru `n` par sau `(n+1)/2` pentru cazul `n` impar (există în mijloc un chenar degenerat, format dintr-un singur element). Chenarul `c` are colțul stânga-sus (c,c) și colțul dreapta jos (`n-1-c`, `n-1-d`). Pentru a calcula suma elementelor de pe un chenar nu este necesar să parcurgem separat fiecare latură, este suficient să parcurgem o singură latură, dar să adunăm patru elemente deodată (câte unul pentru fiecare dintre cele patru laturi). Singura problemă este că în acest mod adunăm colțurile de două ori.

```
for (c=0; c<n/2; c++)
   (for(s=0, i=c; i<n-c; i++)
       s+=a[c][i]+a[i][c]+a[c][n-1-c]+a[n-1-c][i];
    //scad colturile
    s-=a[c][c]+a[n-1-c][c]+a[c][n-1-c]+a[n-1-c][n-1-c];
    cout<<"Suma pentru chenarul "<<c<<" = "<<s<<endl; }
if (n%2) //scriu chenarul din mijloc
   cout<<"Suma pentru chenarul "<<n/2<<" = "<<a[n/2][n/2]<<endl;
```

21. Parcurg fișierul secvențial, calculând în vectorul `l` frecvența de apariție a fiecărei litere. Verific apoi dacă în vectorul `l` există o componentă egală cu `k`.

```
#include <fstream.h>
int l[26], k;
void main()
{ ifstream f("bac2000.txt");
  char c;
  cout<<"k="; cin>>k;
  while (f.get(c))
        if (c!=' ')     //c este litera mare
            l[c-'A']++; //maresc frecventa de aparitie a literei c
  f.close();
  for (int i=0; i<26; i++)
      if (l[i]==k) {cout <<"DA"; return;}
  cout<<"NU"; }
```

23. Vom completa în primul rând ultima linie și ultima coloană cu 1. Apoi vom calcula celelalte elemente în ordine de jos în sus și de la dreapta spre stânga.

```
for (i=0; i<n; i++)  a[i][n-1]=a[n-1][i]=1;
for (i=n-2; i>=0; i--)
    for(j=n-2; j>=0; j--)
       a[i][j]=a[i+1][j]+a[i][j+1];
```

25. Vom reprezenta un polinom ca un vector în care reținem coeficienții săi, în ordinea crescătoare a gradelor. Pentru a fi automat inițializate cu 0, declarăm polinoamele în afara funcției main().

```
#include <iostream.h>
typedef double Polinom[100];
Polinom P, Q, S, M;
int Pgrad, Qgrad, Sgrad, Mgrad; //gradele polinoamelor
int main ()
{int i, gmax, j;
 cout << "\nGradul polinomului P = "; cin >> Pgrad;
 cout<<"\nCoeficientii polinomului P, in ordinea gradelor:";
 for (i=0; i<=Pgrad; i++) cin >> P[i];
 cout << "\nGradul polinomului Q = "; cin >> Qgrad;
 cout<<"\nCoeficientii polinomului Q, in ordinea gradelor:";
 for (i=0; i<=Qgrad; i++) cin >> Q[i];
 gmax = Pgrad>Qgrad ? Pgrad: Qgrad;
 for (i=0;i<=gmax;i++) S[i]=P[i]+Q[i];                    //suma
 for (; gmax && !S[gmax]; gmax--);
 Sgrad=gmax;
 cout<<"\nPolinomul suma are coeficientii ";
 for (i=0; i <= Sgrad; i++) cout << S[i] <<' ';
 Mgrad=Pgrad+Qgrad;                                       //produsul
 for (i=0; i<=Pgrad; i++)
    for (j=0; j<=Qgrad; j++)  M[i+j]+=P[i]*Q[j];
cout<<"\nPolinomul produs are coeficientii ";
for (i=0; i <= Mgrad; i++) cout <<M[i]<<' '; return 0;}
```

26. Ordonăm crescător elevii după înălțime și schiurile după lungime, apoi distribuim schiurile în această ordine.

29. Se sortează crescător elementele vectorului. Dacă numărul de elemente este par, ultimul element se ignoră. Se parcurge prima jumătate a vectorului, afișând alternativ un element din prima jumătate și un element din cea de-a doua. La sfârșit se mai afișează doar mijlocul.

30.

```
#include <fstream.h>
int a[100][100], n, p[100];
int main()
{ ifstream f("relatii.in");
  int o[100], i, j, aux, ok, nr;
  f>>n;           //citesc numarul de elevi si matricea relatiilor
  for (i=0; i<n; i++)
      for (j=0; j<n; j++) f>>a[i][j];
  f.close();
  cout <<"Perechile de elevi care se agreeaza reciproc:"<<endl;
  for (i=0; i<n; i++)
      for (j=i+1; j<n; j++)
          if (a[i][j] && a[j][i]) cout <<i+1<<' '<<j+1<<endl;
  cout<<"Elevii care nu sunt agreati de nimeni:";
  for (i=0; i<n; i++)
      {for(j=0;j<n; j++) //numar cati elevi agreeaza elevul i
         p[i]+=a[j][i];
       if (p[i]==1) cout<<i+1<<' ';}
  cout<<endl;
  cout<<"Elevii care nu agreeaza pe nimeni: ";
  for (i=0; i<n; i++)
      {for(nr=j=0; j<n; j++) //numar elevii agreati de elevul i
         nr+=a[i][j];
```

```
        if (nr==1) cout<<i+1<<' ';}
    cout<<"\nElevii in ordinea descrescatoare a popularitatii:\n";
    //determin ordinea elevilor in functie de popularitate
    for (i=0; i<n; i++) o[i]=i+1;
    do {ok=1;
        for (i=0; i<n-1; i++)
            if (p[o[i]-1]<p[o[i+1]-1])
                {aux=o[i];o[i]=o[i+1];o[i+1]=aux;ok=0;}
        } while (!ok);
    for (i=0; i<n; i++) cout<<o[i]<<' '; cout<<endl; return 0;}
```

31.

```
#include <iostream.h>
int main ()
{char p[200];  int i;
 cin.getline(p,200,'.');                              //citesc propozitia
 for (i=0; p[i]; i++)                                 //parcurg propozitia
     if ('a'<=p[i] && p[i]<='z')                       //este litera mica
        p[i]=p[i]-'a'+'A';                      //o convertesc in majuscula
  cout <<p<<endl; return 0;}                          //afisez propozitia
```

32.

```
#include<iostream.h>
#include<string.h>
int main()
{char c[30], p[1000], *poz=p;
 cin.getline(c, sizeof(c)); cin.getline(p, sizeof(p));
 for (int n=strlen(c), i=0; poz && i<n; i++) poz=strchr(poz,c[i]);
 if (poz) cout<<"Literele cuvantului apar in ordine in propozitie";
    else cout<<"Literele cuvantului nu apar in ordine in propozitie"; }
```

33.

```
#include <iostream.h>
#include <string.h>
void main ()
{ int n, i, j,st, dr, gasit, mijloc, rez;
  struct Persoana {char nume[40];
                   unsigned long tel;} Ag[200], v;
  char x[40];
  cout <<"n="; cin >> n;
cout << "Introduceti informatiile\n";//sortam de la citire, prin insertie
for (i=0; i<n; i++)
    {cout << "Numele si prenumele: ";  cin.get();cin.getline(v.nume,40);
     cout << "Telefonul: "; cin>>v.tel;
     //inserez informatiile citite in agenda, pe pozitia corecta
     for (j=i; j && strcmp (Ag[j-1].nume, v.nume)>0; j--)
         Ag[j]=Ag[j-1];
     Ag[j]=v;}
cout <<"Numele persoanei cautate:";  cin.get();cin.getline(x, 40);
for (st=0,dr=n-1, gasit=0; !gasit && st<=dr; )            //cautare binara
    {mijloc=(st+dr)/2;
     rez= strcmp(Ag[mijloc].nume, x);
     if (!rez) gasit=1;
        else if (rez>0) dr=mijloc-1;
                else st=mijloc+1;}
if (gasit) cout <<"Telefon: "<<Ag[mijloc].tel;
   else cout <<"Persoana "<<x<<" nu exista in agenda"; }
```

34.

```
#include <iostream.h>
#include <string.h>
int main ()
{ char t[1000], c[20], *p=t, *rest;
  cin.getline(c,20);  cin.getline(t,1000);
```

```
do
{ p=strstr(p,c);          //caut c in t, incepand cu pozitia p
  if (p)                  //c apare in t pe pozitia p, il sterg
     { rest=p+strlen(c); *p=NULL;
       strcat(t,rest);}
} while(p);
cout <<t<<endl; return 0;}
```

35. Vom utiliza principiul cutiei al lui Dirichlet: construim numerele 1, 11, 111, 1111, ..., 11...11 (n de 1). Sunt n numere distincte, care au toate cifrele egale cu 1. Calculăm resturile împărțirii numerelor la n. Dacă există un număr pentru care restul este egal cu 0, acela este multiplul căutat. Dacă nu, deducem că cele n resturi iau valori în mulțimea {1, 2, ..., n-1}, deci există două resturi egale. Prin urmare, există două astfel de numere, a căror diferență este divizibilă cu n. Diferența lor este multiplul căutat.

36. Vom spune că o lucrare este *restantă* dacă execuția ei se termină după termenul de predare; altfel, o vom numi *în termen*. Observăm că orice planificare poate fi adusă la o formă echivalentă (din punctul de vedere al penalizării totale) în care toate lucrările *în termen* precedă lucrările *restante*. Mai mult, orice planificare poate fi adusă în *formă canonică*, astfel încât lucrările *în termen* să preceadă lucrările *restante*, iar lucrările *în termen* să fie în ordinea crescătoare a termenelor de predare (dacă într-o planificare oarecare L, lucrarea în termen L_i are termenul de predare ulterior termenului de predare a lucrării în termen L_j, i<j, putem inversa în planificare execuția lucrărilor L_i și L_j, acestea rămânând în termen). Deci problema determinării unei planificări optimale se reduce la a determina lucrările în termen în ordinea crescătoare a termenelor de predare. Vom ordona lucrările descrescător după penalizări și vom construi mulțimea L a lucrărilor în termen selectând la fiecare pas prima lucrare (deci cea a cărei întârziere ar produce penalizarea cea mai mare), care poate fi executată în termen până la momentul curent, inserând în planificare lucrarea astfel încât lucrările să fie în ordinea crescătoare a termenelor de predare. Utilizăm următoarele variabile:

```
int  n, nt, nr;
//n - nr. de lucrari;
// nt - nr. de lucrari "in termen";
//nr - numarul de restante
int t[NMax];           //termenele de predare a lucrarilor
double p[NMax];        //penalizarile pentru restante
int o[NMax];          //ordinea lucrarilor in functie de penalizari
int R[LNMax]         //multimea lucrarilor restante
int L[NMax];        //multimea lucrarilor "in termen"
```

După citirea și ordonarea lucrărilor descrescător după penalizări, determinăm mulțimea lucrărilor în termen astfel:

```
nt=1; L[0]=o[0];
for (i=1;i<n; i++)
    {//incerc sa inserez lucrarea o[i] in L
    for (j=nt-1;j>=0 && t[L[j]]>(j+1) && t[L[j]] > t[o[i]]; j--);
     //Lucrarea L[j] nu mai poate fi decalata
     j++;
     //verific daca lucrarea o[i] ar fi "in termen" la momentul j
 if (t[o[i]] > j)
    {
       /* inserez lucrarea o[i] in L pe pozitia j,
          decaland celelalte lucrari */
        for (k=nt++;k>j;k--) L[k]=L[k-1];          //am decalat
        L[j]=o[i];}
        else                                  //lucrare restanta
        {ptotal+=p[o[i]]; R[nr++]=o[i];}
    }
```

38. Metoda de programare utilizată este *Greedy*. Cât timp mai există ore de plecare „nerezolvate": punem în funcțiune un nou bac, sortăm timpii de plecare, alegem timpul minim (acesta va reprezenta prima cursă pentru acest bac); simulăm toate cursele pe care le poate rezolva acest bac.

39. Pentru fiecare persoană trebuie să reținem două liste: una formată din persoanele despre care ea

afirmă că spun adevărul, cealaltă din persoanele despre care afirmă că mint. Pentru aceasta vom utiliza două matrice cu n linii şi n coloane. În matricea A vom reţine pe linia i lista persoanelor despre care persoana i afirmă că spun adevărul. În matricea F vom reţine pe linia i lista persoanelor despre care persoana i afirmă că mint.

Observăm că în matricele A şi F diferă numerele de elemente de pe linii.

Deci vom utiliza şi doi vectori, nr_a şi nr_f, astfel:

nr_a[i] = numărul de persoane despre care persoana i afirmă că spun adevărul

nr_f[i]= numărul de persoane despre care persoana i afirmă că mint.

Pentru a reprezenta o soluţie vom utiliza un vector cu n componente denumit stare, cu semnificaţia: stare[i]=0 dacă persoana i spune adevărul şi 1 dacă persoana i minte. În vectorul stare vom construi succesiv toate configuraţiile de 0 şi 1 până la depistarea unei configuraţii care respectă afirmaţiile memorate în A şi F. Pentru aceasta vom iniţializa vectorul stare cu 0 şi vom aduna 1 în baza 2 până când configuraţia este corectă.

40.

```
#include <fstream.h>
#define MAX_N 100
int main(void)
{int i, j, n, m;
 double timp[MAX_N], best_timp, dif;
  int clasament[MAX_N], temp;
  ifstream fisin("schi.in");
  fisin>>n>>m;
  fisin>>timp[0];   //citesc timpul primului concurent din runda 1
  best_timp = timp[0];   //deocamdata acesta este cel mai bun timp
  for (i = 1; i < n; ++i) //pentru fiecare concurent urmator
    {
     //citesc diferenta de timp fata de cel mai bun timp
     fisin>>dif;
     timp[i] = best_timp + dif;     //calculez timpul efectiv
     if (dif < 0)                   //daca a scos un timp mai bun
       best_timp = timp[i];        //acesta devine cel mai bun timp
    }
  for (i = 0; i < n; ++i)   //fac clasamentul dupa prima runda
    clasament[i] = i;
  for (i = 0; i < n - 1; ++i)
    for (j = i + 1 ;j < n; ++j)
      if (timp[clasament[i]] > timp[clasament[j]])
          {temp = clasament[i];
           clasament[i] = clasament[j];
           clasament[j] = temp;      }
//citesc timpul celui de al m-lea concurent
fisin>>timp[clasament[m-1]];
best_timp = timp[clasament[m - 1]];
//deocamdata acesta este cel mai bun timp total
for (i = m - 2; i >= 0; --i)
  { fisin>>dif;//citesc diferenta de timp fata de cel mai bun timp
    //calculez timpul total efectiv
    timp[clasament[i]]=best_timp+dif;
    if (dif < 0)                //daca a scos un timp total mai bun
       //acesta devine cel mai bun timp total
       best_timp = timp[clasament[i]];
  }
for (i = 0; i < m - 1; ++i)   //fac clasamentul dupa runda a doua
   for (j = i + 1; j < m; ++j)
      if (timp[clasament[i]] > timp[clasament[j]])
         { temp = clasament[i];
           clasament[i] = clasament[j];
           clasament[j] = temp;          }
```

```
fisin.close();
  ofstream fisout("schi.out");      //afisez primii 3 clasati
  fisout << clasament[0] + 1 << endl;
  fisout << clasament[1] + 1 << endl;
  fisout << clasament[2] + 1 <<endl;
  fisout.close();  return 0; }
```

5. Stiva şi coada

1. a. *Fals* (coada este o structură de date abstractă, nu un vector; ea poate fi implementată static, cu ajutorul unui vector); **b.** *Adevărat*; **c.** *Fals* (coada este o structură de date care funcționează după principiul FIFO, stiva funcționează după principiul LIFO); **d.** *Fals* (pentru a afișa elementele unei stive trebuie să extragem succesiv elementele din stivă, deoarece nu avem acces direct decât la elementul din vârful stivei). **4.** Pentru a face distincția între cazul în care coada este vidă (nu conține nici un element) și cazul în care coada este plină (toată zona de memorie alocată cozii este ocupată), vom reține, pe lângă începutul și sfârșitul cozii și `Nr`, numărul de elemente din coadă.

a. Inserarea elementului `x` în coada `C`:

```
if (Nr == DimMaxCoada) {cout << "Eroare "; return;}
    else {Sf=(Sf+1)%DimMaxCoada; C[Sf]=x; Nr++;}
```

b. Extragerea unui element:

```
if (!Nr) {cout << "Eroare "; return;}
 else {x = C[Inc]; Inc = (Inc+1)% DimMaxCoada; Nr--;};
```

5. Deoarece primul pacient intrat va fi primul pacient tratat, funcționarea cabinetului poate fi simulată cu ajutorul unei cozi, în care vom înregistra pacienții în ordinea sosirii lor. Când medicul devine liber, un pacient este extras din coadă și tratat. La fiecare minut vom testa dacă se produce un eveniment (vin pacienți sau/și medicul termină tratamentul unui pacient). În funcție de evenimentul produs se inserează un element în coadă (pacientul nou sosit) sau se afișează timpul la care pleacă pacientul tratat și se extrage un element din coadă (dacă există). **6.** Liniile de intrare, cele `k` linii de manevră și linia de ieșire funcționează, în fapt, ca structuri de tip coadă, pe care le vom implementa cu ajutorul vectorilor. Pe cele `k` linii de manevră, care pot conține câte maxim `n` vagoane, sunt dispuse, în ordine, vagoanele care intră de pe linia de intrare. Pentru fiecare vagon care intră de pe linia de intrare se verifică dacă acesta poate intra pe una dintre liniile de manevră. Pentru aceasta se verifică pe rând liniile de manevră. Vagonul poate intra pe una dintre ele dacă fie linia de manevră este goală, fie numărul vagonului este cu `1` mai mare decât ultimul vagon deja ajuns pe linia de manevră respectivă. Dacă vagonul se poate muta pe una dintre liniile de manevră, se realizează mutarea urmată imediat de verificarea posibilității de mutare a unor vagoane de pe liniile de manevră pe linia de ieșire, adică operația de extragere a unor elemente din coadă. **7.** Problema este asemănătoare cu problema caroiajului, fiind cerută obținerea unui drum de lungime minimă. De data aceasta nu mai există obstacole, trecerea din poziția curentă într-o poziție învecinată (sus, jos, stânga, dreapta) realizându-se în funcție de valorile acestora. Prin urmare, vom utiliza o coadă în care vom memora pozițiile din labirint în ordinea în care acestea sunt parcurse. Inițial în coadă plasăm poziția de plecare – `(1,1)`. La fiecare pas extragem un element din coadă, după care inserăm în coadă toate pozițiile vecine cu poziția elementului extras care respectă condiția de deplasare. Algoritmul se termină când coada devine vidă sau când am atins destinația – poziția `(n,m)`. Deoarece problema cere și drumul minim parcurs, vom reține pentru fiecare poziție din labirint atinsă în timpul acestei parcurgeri și direcția din care a fost atinsă (caracterul `D` – jos, `U` – sus, `L` – stânga, respectiv `R` – dreapta).

8. Problema este asemănătoare cu problema caroiajului, numai că trebuie să o rezolvăm de două ori: o dată pentru a determina timpul minim în care ajunge Julieta în fiecare poziție, respectiv cel în care ajunge Romeo. Vom determina o poziție în care cei doi ajung în același timp minim.

6. Funcții

1. a. *Fals* (la apel intervin parametrii actuali); **b.** *Fals* (se specifică doar numele parametrilor actuali); **c.** *Adevărat*; **d.** *Adevărat*; **e.** *Fals* (lista de parametri formali poate să fie vidă).
2. c, e. **3.** b. **4.** a. **5.** b. **6.** Variabila locală `nr` nu este inițializată, funcția nu returnează nici o valoare, pentru `n=0` `nr` este 0. **7.** 611215112122232351123. **8.** c. **9.** 133233. **10.** c.
11. Pentru calculul expresiei se scriu două funcții utilizator, `Min` și `Max`, pentru determinarea minimului, respectiv maximului dintre două numere reale. Calcularea valorii expresiei E se face apoi într-o instrucțiune alternativă imbricată. Pentru determinarea valorii absolute a unui număr real se utilizează funcția `abs`. **12.** Utilizând cunoștințele de la matematică cu privire la operațiile cu numere complexe, se scrie câte o funcție pentru citirea unui număr complex, afișarea unui număr complex, determinarea modulului unui număr complex, suma, diferența, produsul a două numere complexe.
13. Reprezentăm vârfurile triunghiului ca o structură cu două câmpuri – coordonatele `x` și `y` ale punctului respectiv. Pentru calcularea diferitelor elemente vom utiliza trei funcții: distanța dintre două puncte (`Latura`), aria triunghiului (`S`), raza cercului circumscris (`R`).

```
struct Punct {float  x, y;} Pa, Pb, Pc;
float Latura(Punct X, Y)
{  return sqrt((Y.x-X.x)* (Y.x-X.x)+(Y.y-X.y)* (Y.y-X.y));}
float S(float a, float b, float c)
{ float p=(a+b+c)/2;                                    //semiperimetrul
  return sqrt(p*(p-a)*(p-b)*(p-c));  }                  //formula lui Heron
float R(Punct Pa, Punct Pb, Punct Pc)
{ float a, b, c;
  a = Latura(Pb, Pc); b = Latura(Pa, Pc); c = Latura(Pa, Pb);
  return a*b*c/(4*S(a, b, c));    }                     //raza cercului circumscris
```

14. a.

```
void elimin0(int a[20], int & n)
{ int nr=0, i;
  //nr este numarul de zerouri gasite pana la un moment dat in vector
  for (i=0; i<n; i++)
      if (!a[i]) nr++;                                  //am mai gasit un zero
         else                                           //am gasit un element nenul
         if (nr)                                        //am gasit zerouri
         {//plasez elementul nenul pe pozitia primului zero din vector
            a[i-nr]=a[i];
            a[i]=0; }
n-=nr;  }        //n indica acum numarul elementelor nenule din vector
```

b. Singura modificare în funcția șablon este antetul:

```
template <class T>
void elimin0(T a[20], int & n)
```

15. O soluție ar fi de a reprezenta funcția ca o matrice cu elemente 0 și 1: `Mat[i][j]=1` dacă și numai dacă `f(i)=j`. Acest mod de reprezentare prezintă avantajul de a putea verifica foarte ușor următoarele:

- dacă funcția este bine definită – matricea definește o funcție dacă fiecare element din domeniul de definiție are exact o imagine, adică suma elementelor de pe fiecare linie trebuie să fie egală cu 1;
- dacă funcția este injectivă – funcția este injectivă dacă fiecare element din codomeniu are cel mult o imagine, adică suma elementelor de pe fiecare coloană trebuie să fie cel mult egală cu 1;
- dacă funcția este surjectivă – funcția este surjectivă dacă fiecare element din codomeniu are cel puțin o imagine, adică suma elementelor de pe fiecare coloană trebuie să fie cel puțin egală cu 1.

O altă soluție ar fi de a reprezenta funcția ca un vector în care reținem pentru fiecare element din domeniu imaginea sa. Utilizând această reprezentare, funcția este întotdeauna bine definită. Pentru a testa injectivitatea verificăm dacă oricare două componente ale vectorului sunt diferite. Pentru a testa surjectivitatea verificăm dacă mulțimea componentelor vectorului (imaginea funcției) coincide cu domeniul funcției.

16. a.

```
char* inserare(char* s2, char* s1, int n)
{ int i, lg1=LgSir(s1), lg2=LgSir(s2);
 //deplasez caracterele de la n la lg2 cu lg1 pozitii spre dreapta
 for (i=lg2; i>=n; i--) s2[i+lg1]=s2[i];
 //inserez s1 in locul disponibil
 for(i=0; s1[i]; i++) s2[i+n]=s1[i]; return s2; }
```

b. Deplasez în șirul de caractere dat caracterele de după poziția `n+nr` cu `nr` poziții la stânga.

17. Pentru a reprezenta prin vectorul caracteristic o mulțime de maxim `DimMax` elemente, vom utiliza un vector cu `DimMax/8` componente (putem presupune fără a restrânge generalitatea că `DimMax` este multiplu de 8, altfel îl înlocuim cu cel mai mic multiplu al lui 8, mai mare decât `DimMax`). Elementului cu numărul de ordine `i` ($i \in \{0,1,\ldots,DimMax-1\}$) îi corespunde bitul `i%8` din octetul `i/8`. Descriem principalele operații cu mulțimi, utilizând această reprezentare.

```
#define DimMax 1024
#define Lg (DimMax/8)
typedef unsigned char Multime[Lg];
int in (unsigned x, Multime M)
{   //intoarce 1 daca x apartine multimii M si 0 altfel
    return M[x/8] >> x%8 & 1; }
int inclus (Multime M1, Multime M2)
{   //returneaza 1 daca M1 inclus sau egal cu M2 si 0 altfel
    for (int i=0; i < DimMax; i++)
        if((M1[i/8] >> i%8 & 1)>(M2[i/8] >> i%8 & 1)) return 0;
    return 1;  }
void adauga(unsigned x, Multime M)
{    //insereaza elementul x in multimea M
     M[x/8]|=(1<< x%8);  }
void extrage(unsigned x, Multime M)
{    //elimina elementul x din multimea M
     M[x/8]&=~(1<< x%8);   }
void reuniune (Multime M1, Multime M2, Multime M)
{    //rezultatul reuniunii va fi in M
     for (int i=0; i<Lg;i++) M[i] = M1[i] | M2[i]; }
void intersectie (Multime M1, Multime M2, Multime M)
{   //rezultatul intersectiei va fi in M
    for (int i=0; i<Lg;i++) M[i] = M1[i] & M2[i];   }
void diferenta (Multime M1, Multime M2, Multime M)
{   //rezultatul diferentei M1-M2 va fi in M
    for (int i=0; i<Lg;i++) M[i] = M1[i] & ~M2[i];  }
void diferentaS (Multime M1, Multime M2, Multime M)
{   //rezultatul diferentei simetrice va fi in M
    Multime D1, D2;
    diferenta (M1, M2, D1); diferenta (M2, M1, D2);
    reuniune (D1, D2, M);                          }
void afisare(Multime M)
{    for (unsigned i=0; i<DimMax; i++)
         if (in(i, M)) cout << i<< ' ';
     cout << endl;                                 }
void citire(Multime M)
{    int cate;   unsigned x;
     for (unsigned i=0; i<Lg; i++) M[i]=0;
     cout <<"Cate elemente are multimea ? "; cin>>cate;
     cout <<"Introduceti elementele multimii ";
     for (i=0; i<cate; i++) {cin >> x; adauga(x, M);}
     cout << endl;                                 }
```

18. Vom crea un fişier antet care va conţine declaraţiile funcţiilor, tipurilor şi constantelor simbolice necesare pentru lucrul cu mulţimi (definite în fişierul `multimi.cpp`), denumit `multimi.h`:

```
#define DimMax 256
#define Lg (DimMax/8)
typedef unsigned char Multime[Lg];
int in (unsigned x, Multime & M);
int inclus (const Multime & M1, const Multime & M2);
void adauga(unsigned x, Multime & M);
void extrage(unsigned x, Multime & M);
void reuniune (Multime & M1, Multime & M2, Multime & M);
void intersectie (Multime & M1, Multime & M2, Multime & M);
void diferenta (Multime & M1, Multime & M2, Multime & M);
void diferentaS (Multime & M1, Multime & M2, Multime & M);
void afisare(Multime & M);
void citire(Multime & M);
```

Apoi vom crea un proiect care conţine fişierul `multimi.cpp` şi următorul fişier care generează numerele prime, prin metoda ciurului lui Eratostene:

```
#include <iostream.h>
#include "multimi.h"
Multime P;
void main()
{ unsigned i, k;
  //plasam in ciur toate numerele naturale pana la DimMax:
  for (i=2; i < DimMax;i++) adauga(i, P);
  for (i=2; i < DimMax/2;i++)
       if (in (i, P))                    //i este in ciur, deci este prim
          for (k=2; k*i<DimMax; k++) //elimin din ciur multiplii lui i
             extrage(k*i,P);
   cout << "Numerele prime sunt \n"; afisare(P);                    }
```

19. Parcurg fişierul de tradus linie cu linie. Traduc linia citită şi o scriu în fişierul de ieşire. Pentru traducerea unei linii extrag succesiv câte un cuvânt şi îi caut traducerea în dicţionar. Dacă am găsit cuvântul în dicţionar, plasez în linia tradusă traducerea cuvântului, altfel scriu cuvântul original. Separatorii îi copiez în linia tradusă aşa cum se află pe linia de tradus.

```
#include <fstream.h>
#include <string.h>
ifstream f, d; ofstream ft;
char linie [100], liniet[100], nume[40];
void Init();
char * Traduc (char *);
void CautDict(char *, char *);
void main ()
{ Init();
  while (!f.eof())       //parcurg fisierul de tradus linie cu linie
       { f.getline(linie, 100);        //citesc o linie din fisier
         ft<<Traduc(linie)<<endl;  }    //traduc linia si o scriu in ft
  f.close(); ft.close();   }
void Init()
{ cout <<"Fisierul care trebuie tradus "; cin.getline(nume, 40);
  f.open(nume);
  cout <<"Fisierul care contine traducerea "; cin.getline(nume, 40);
  ft.open(nume);
  cout <<"Dictionarul "; cin.getline(nume, 40); }
char * Traduc(char linie[100])
{ char sep[]=" ,.;:";                          //contine separatorii
  char cuvant[20], ctradus[20];
  int i, j, k;
                          //extrag succesiv cuvintele de pe linie
  for (i=j=0; i < strlen(linie);)
      { while (i<strlen(linie) && strchr(sep, linie[i]))
              liniet[j++]=linie[i++];          //copiez separatorii
```

```
        //urmeaza un cuvant, il extrag
        for (k=0; i<strlen(linie) &&!strchr(sep, linie[i]);)
            cuvant[k++]=linie[i++];
        cuvant[k++]=NULL;
        CautDict(cuvant, ctradus);              //traduc cuvantul extras
                                                //copiez cuvantul tradus
        for (k=0; ctradus[k]; liniet[j++]=ctradus[k++]);      }
  liniet[j++]=NULL; return liniet; }
void CautDict(char c[20], char ct[20])
{ char l[42], *p;
  d.open(nume);
  while (!d.eof())
        { d.getline(l, 42);          //citesc o linie din dictionar
          p=l;                      // extrag cuvantul si traducerea sa
          if (!strcmp(strtok(p, "="), c))         //am gasit cuvantul
             {  strcpy(ct,strtok(NULL, "\n"));   //copiez traducerea
                d.close();
                return;  }
         }
  strcpy(ct, c);                                 //cuvantul nu se traduce
  d.close(); }
```

20. Vom reține configurația populației într-o matrice `Pop1`, cu `n` linii și `m` coloane. La fiecare pas vom apela funcția `Urmator()`, care va construi, pe baza configurației curente, configurația următoare în matricea `Pop2`. La pasul următor, aceasta va deveni „configurație curentă" (`Pop2` devine `Pop1`). Funcția se apelează repetat, până la apăsarea unei taste.

```
int Pop1[100][100], Pop2[100][100], n, m;
void Urmator()
{int i,j, nv;
for (i=1;i<=n; i++)
    { for (j=1; j<=m; j++)    .
          {                                  //calculez numarul de vecini
            nv=Pop1[i-1][j-1]+Pop1[i-1][j]+Pop1[i-1][j+1]+Pop1[i][j-1]+
               Pop1[i][j+1]+Pop1[i+1][j-1]+Pop1[i+1][j]+Pop1[i+1][j+1];
            switch (nv)
                  {case 2: Pop2[i][j] = Pop1[i][j];
                           if (Pop1[i][j]) cout << '*'; else cout<<' ';
                           break;
                   case 3:  Pop2[i][j] = 1; cout <<'*'; break;
                   default: Pop2[i][j] = 0; cout << ' ';        }
          }
      cout << endl;  }
for (i=1; i<=n;i++)   //configuratia obtinuta devine configuratie curenta
    for (j=1;j<=m; j++) Pop1[i][j]=Pop2[i][j]; }
```

21. Funcția `apare(x,y)` determină de câte ori apare `x` ca subsecvență de cifre în `y`. Funcția va fi apelată în `main()` pentru fiecare număr din intervalul [a, b].

```
unsigned long apare(unsigned long x, unsigned long y)
{                   //determina de cate ori apare x ca subsecventa in y
  char Sx[11], Sy[11], *p, *pSy;
  unsigned long cate = 0;
  ultoa(x, Sx, 10); ultoa(y, Sy, 10); pSy=Sy;
  while ((p=strstr(pSy, Sx))!=NULL)   //x apare in y incepand cu pozitia p
        {  cate++;   pSy=p+1;  }
  return cate; }
```

22. Se va descrie o funcție pentru fiecare metodă de sortare învățată. La apelul unei funcții de sortare se estimează timpul de execuție și se realizează o reprezentare grafică (cu funcții din `graphics.h`).

23. Pentru fiecare număr x_i se determină cifra maximă și bazele în care poate fi considerat că este reprezentat (`cifra maxima+1,...,10`). Pentru fiecare bază posibilă, se convertește numărul x_i în

baza 10, obținându-se astfel pentru fiecare x_i o mulțime de valori v_i. Se calculează produsul cartezian $v_1 \times v_2 \times \ldots \times v_n$ și se alege elementul care determină un interval de lungime minimă.

24. Pentru a determina ordinea în care au fost cusute peticele, le vom „descoase" în ordinea inversă coaserii lor. Mai exact, la fiecare pas vom determina un petic care este deasupra (peste el nu este plasat nici un alt petic) și îl vom descoase (vom înlocui culoarea lui cu alb).

25.

```
#include <stdio.h>
#define MaxM 105
#define MaxN 1005
#define Infinit 1000
#define INFILE "pepsi.in"
#define OUTFILE "pepsi.out"
int tip[MaxN];  /*tip[i]=1 daca persoana i este risipitoare; 0 - altfel */
int volum[MaxM];/*volum[i]=volumul existent la un moment dat in sticla i*/
int final[MaxM];/*final[i]=volumul existent in final in sticla i */
int sol[MaxN];
int perm[MaxM], uz[MaxM];
int M, N, K;
void afisare( void )
{int i;
 FILE *fout = fopen( OUTFILE, "wt" );
 for (i = 1; i <= N; i++ ) fprintf( fout, "%d ", perm[sol[i]] );
 fclose(fout);}
void citire(void)
{int i;
 FILE *fin=fopen( INFILE, "rt" );
 char s[MaxN+1];
 fscanf( fin, "%d %d %d", &N, &M, &K );
 for ( i = 1; i <= M; i++ ) volum[i] = K;
 fscanf( fin, "%s", s );
 for( i = 1; i <= N; i++ )
      if( s[i] == 'R' ) tip[i] = 1; else tip[i] = 0;
 for( i = 1; i <= M; i++ ) fscanf( fin,"%d", &final[i] );
 fclose( fin );}
int main( void )
{int i, j, m, poz;
 citire();
 /*servesc persoanele in ordine */
 for (i = 1; i <= N; i++ )
     if (tip[i] == 1)
         {m = -1; /*determin volumul maxim */
         for (j = 1; j <= M; j++ )
              if (volum[j] > m )
                 {m = volum[j]; poz = j;  }
             sol[i] = poz; volum[poz]--;          }
        else
        {   m = Infinit; /*determin volumul minim */
            for ( j = 1; j <= M; j++)
                if ( volum[j] < m  && volum[j] > 0 )
                   {m = volum[j]; poz = j;    }
            sol[i] = poz;  volum[poz]--;      }
/*in solutia obtinuta este posibil sa fie o alta ordine a sticlelor*/
for (i = 1; i <= M; i++ )
    {j = 1; /*determin renumerotarea sticlelor */
     while (final[i] != volum[j] || uz[j]) j++;
     perm[j] = i; uz[j] = 1;     }
afisare(); return 0; }
```

26. Citim succesiv operațiile și pentru fiecare determinăm tipul și apelăm funcția corespunzătoare (determinare număr submulțime `detnr()` sau determinare submulțime `detsub()`).

```
long detnr(void)
{long nro=0, gasit=0;
 int pas;
 for (pas=1; pas<=n && gasit<x; pas++)
      if (!s[pas]) nro+=1l<<n-pas;
         else
         {nro++;gasit++;}
return nro;}
void detsubm(void)
{int pas=0;
 while (nr)
      {pas++;
       if (nr<=1l<<n-pas) {s[pas]=1; nr--;}
         else {nr-=1l<<n-pas;}
      }
}
```

27. Orice număr natural are o scriere unică sub formă de sumă de termeni neconsecutivi ai şirului Fibonacci (demonstraţia se poate face prin inducţie). Vom determina cel mai mare număr Fibonacci care este mai mic sau egal cu N, îl afişăm, apoi îl scădem din N şi reluăm algoritmul pentru noul N. Sunt necesare operaţii pe numere mari. **28.** Ordonăm butoaiele crescător după capacitate. Calculăm volumul total de apă acumulat în vase, apoi îl distribuim în ordine în butoaie. Atenţie, trebuie să utilizaţi operaţii cu numere mari.

29.

```
#include <stdio.h>
#define INPUTFILE "masina.in"
#define OUTPUTFILE "masina.out"
#define MAXR 50
#define MAXS 50
#define Culoar '.'
#define Zid 'X'
#define OK '*'
#define MaxDir 1000
int n, r, s, InitL, InitC, LgC, LgAux;
char harta[MAXR][MAXS+1];
typedef struct { int r, s; } Deplasare;
Deplasare Dir[MaxDir];
Deplasare Coada[MAXR*MAXS];
Deplasare AuxC[MAXR*MAXS];
void citire(void)
{int i, j;
 char str[20];
 FILE *f = fopen(INPUTFILE, "rt");
 fscanf(f, "%d%d", &r, &s);
 for (i=0; i<r; i++)
     {fscanf(f, "%s", harta[i]);
      for (j=0; j<s; j++)
          if (harta[i][j]==OK) //retin pozitia initiala a masinii
               {InitL = i; InitC = j;}
      }
 fscanf(f, "%d", &n);
 for (i=0; i<n; i++)
      {fscanf(f, "%s", str);
       if (str[0]=='N') { Dir[i].r = -1; Dir[i].s = 0; }
       if (str[0]=='S') { Dir[i].r = 1; Dir[i].s = 0; }
       if (str[0]=='V') { Dir[i].r = 0; Dir[i].s = -1; }
       if (str[0]=='E') { Dir[i].r = 0; Dir[i].s = 1; }
      }
 fclose(f); }
```

```
void rezolva(void)
{ int p, i, tr, ts;
  LgC = 1;  //initializez coada cu pozitia initiala a masinii
  Coada[0].r = InitL; Coada[0].s = InitC;
  for (p=0; p<n; p++)
      {for (i=0; i<LgC; i++) harta[Coada[i].r][Coada[i].s] = Culoar;
      LgAux = 0;
      for (i=0; i<LgC; i++)
          {tr = Coada[i].r; ts = Coada[i].s;
           while (1)
                { tr += Dir[p].r; ts += Dir[p].s;
                     if (tr<0 || tr>=r || ts<0 || ts>=s) break;
                     if (harta[tr][ts]!=Culoar) break;
                     AuxC[LgAux].r = tr; AuxC[LgAux++].s = ts;
                     harta[tr][ts] = OK; };
          }
      for (i=0; i<LgAux; i++) Coada[i] = AuxC[i];
      LgC = LgAux;   }
}
void afisare(void)
{int i;
 FILE *f = fopen(OUTPUTFILE, "wt");
 for (i=0; i<r; i++) fprintf(f, "%s\n", harta[i]);
 fclose(f);   }
int main()
{     citire(); rezolva();   afisare(); return 0;    }
```

Bibliografie

Andonie, Razvan; Gârbacea, Ilie, *Algoritmi fundamentali. O perspectiva C++*, Editura Libris, Cluj-Napoca, 1995.

Atanasiu, A., *Cum se scrie un algoritm? Simplu*, Editura Agni, Bucureşti, 1993.

Atanasiu, Adrian; Pintea, Rodica, *Culegere de probleme Pascal*, Editura Petrion, Bucuresti, 1996.

Bostan, Gheorghe, *Culegere de probleme de informatică*, Editura Lumina, Chişinău, 1999.

Catrina, Octavian; Cojocaru, Iulian, *Turbo C++*, Editura Teora, Bucureşti, 1993.

Cerchez, Emanuela; Şerban, Marinel, *Informatică. Manual pentru clasa a IX-a*, Editura Didactică şi Pedagogică, Bucureşti, 2004.

Cerchez, Emanuela; Şerban, Marinel, *Informatică. Varianta Pascal. Manual pentru clasa a X-a*, Editura Polirom, Iaşi, 2000.

Cerchez, Emanuela, *Informatică. Culegere de probleme pentru liceu*, Editura Polirom, Iaşi, 2002.

Cormen, Thomas; Leiserson, Charles; Rivest, Ronald, *Introduction to Algorithms*, The Massachusetts Institute of Technology, 1990.

Horowitz, Ellis; Sahni, Sartaj; Anderson-Freed, Susan, *Fundamentals of Data Structures in C*, Computer Science Press, New York, 1993.

Ivaşc, Cornelia; Prună, Mona; Mateescu-Cerchez, Emanuela, *Bazele Informaticii. Caiet de laborator*, Editura Petrion, Bucureşti, 1997.

Jamsa, Kris, *Succes cu C++*, Editura All, Bucureşti, 1997.

Jamsa, Kris, *Totul despre C şi C++*, Editura Teora, Bucureşti, 2001.

Livovschi, Leon; Georgescu, Horia, *Sinteza si analiza algoritmilor*, Editura Stiintifica si Enciclopedica, Bucuresti, 1986.

Lucanu, Dorel, *Bazele proiectarii programelor si algoritmilor*, Editura Universitatii „Al.I. Cuza", Iasi, 1996.

Mitrana, Victor, *Provocarea algoritmilor*, Editura Agni, Bucuresti, 1994.

Negrescu, Liviu, *Limbajul C*, Editura Computer Libris Agora, Cluj-Napoca, 1997.

Niculescu, St.; Cerchez, Em.; Şerban M. ş.a.; *Bacalaureat şi atestat în informatică*, Editura L&S Infomat, Bucureşti, 2000.

Sedgewick, Robert, *Algorithms in C++*, Addison-Wesley Publishing Company, Boston, MA, 1992.

S.N.E.E., Subiecte Bacalaureat 2000-2003.

microinformatica

Editat sub egida
Comisiei Naţionale de Informatică

INIŢIERE ÎN C++
PROGRAMARE ORIENTATĂ PE OBIECTE

(Ediţie revizuită)

Cluj-Napoca
1993

Autor
Muşlea Ionuţ

Redactor
Poenaru Codruţa

Designer
Derveşteanu Liviu

Editor
prof. univ. Muntean Emil

Tehnoredactare computerizată
Poenaru Codruţa

© Copyright MicroInformatica SRL, 1993

CUPRINS

CUVÎNT ÎNAINTE 7

1. INTRODUCERE ÎN O.O.P. 9
1.1. Scopul O.O.P. 9
1.2. Scurt istoric 10
1.3. Avantaje O.O.P. 12
1.4. Înainte de început 13

2. SCURTĂ INCURSIUNE ÎN C++ 15
2.1. Intrări şi ieşiri. Stream-uri 15
2.2. Funcţii cu parametrii impliciţi. Funcţii cu acelaşi nume . . 16
2.3. Referinţa 19
2.4. NEW şi DELETE 24
2.5. Clase 25
2.6. Constructori şi destructori 30
2.7. "Redefinirea"operatorilor 35
2.8. FRIENDS 37
2.9. Clase derivate 39
2.10. Polimorfism 45

3. CLASE 48
3.1. Introducere 48
3.2. Declararea claselor 48
3.3. Membrii unei clase. "Scope access operator" 52
3.4. Accesarea membrilor unei clase. Pointeri la metode 58
3.5. Alte amănunte privind declararea claselor. Funcţii "inline" . 62
3.6. Membrii statici 68
3.7. Constructori 71
3.7.1. Iniţializarea structurilor de date 71
3.7.2. Proprietăţi specifice constructorilor 74
3.7.3. Tipuri de constructori. Ambiguităţi 76
3.7.4. Conversii 79
3.7.5. Apelul explicit al constructorilor 80
3.7.6. Reducerea numărului de constructori 82
3.7.7. Constructori şi obiecte statice 82
3.7.8. Constructorii claselor încuibărite 83

3.8. Destructori 87
3.8.1. Definire şi caracteristici 87
3.8.2. Destructorii claselor încuibărite 89
3.8.3. Erori deosebite 90
3.9. Friends 91
3.10. Comportarea constructorilor şi destructorilor în situaţii necesitînd alocare dinamică de memorie 95
3.10.1. Crearea şi distrugerea şirurilor de obiecte 95
3.10.2. "Scurtcircuitarea" mecanismului implicit de gestionare a memoriei alocate dinamic 99
3.11. Probleme rezolvate. Probleme propuse spre rezolvare . . . 109
3.11.1. Array-uri de dimensiuni variabile 109
3.11.2. Hash-table pentru numere de telefon 113
3.11.3. Mecanism perfecţionat de gestiune a memoriei . . . 118

4. MOŞTENIREA 127
4.1. Introducere 127
4.2. Clase derivate 128
4.3. Drepturi de accesare a membrilor "moşteniţi" 130
4.4. Constructori şi destructori 138
4.5. Metode virtuale. Utilizarea listelor eterogene 145
4.6. Moştenire multiplă 157
4.7. Problemă rezolvată. Exerciţii propuse spre rezolvare 167

5. ALTE FACILITĂŢI ALE LIMBAJULUI C++ 178
5.1. Funcţii pure şi clase abstracte 178
5.2. Overloading Operators 180
5.3. Operatori unari şi binari 182
5.4. Operatorii "new" şi "delete" 190
5.5. Operatorul "()" 193
5.6. Operatorul "=" 196
5.7. Operatorul "[]" 201
5.8. Operatorul "->" 204
5.9. Conversii 206
5.10. Problemă propusă spre rezolvare 211

6. STREAM-URI 213
6.1. Introducere 213

6.2. Operaţii de intrare şi ieşire 215
6.2.1. Introducere 215
6.2.2. STREAM-uri şi tipuri de date predefinite 216
6.2.2.1. STREAM-uri de intrare 216
6.2.2.2. STREAM-uri de ieşire 218
6.2.3. Alte metode de manipulare a STREAM-urilor 219
6.2.4. Extractori şi insertori definiţi de utilizator 222
6.3. Tratarea erorilor apărute în operaţii cu stream-uri 224
6.4. Operaţii de intrare şi ieşire cu format 226
6.4.1. Indicatorii de stare ai formatului 226
6.4.2. Manipulatori 229
6.5. Fişiere şi stream-uri 232
6.6. Probleme propuse spre rezolvare 236

ANEXA A 237
Lumea minunată a OOP 237

BIBLIOGRAFIE 240

Pentru Codruţa

"Obiectele sînt reprezentări naturale care facilitează înţelegerea şi modificarea unui program."
Joseph Weizenbaum

CUVÎNT ÎNAINTE

Deşi teoria programării orientate pe obiecte era bine fundamentată încă de acum 20 de ani, ideea nu a prins cu adevărat decît în ultimii ani. Ce se ascunde în spatele conceptului de programare obiectuală? Ideea de a oferi programatorului un instrument performant de reprezentare a lumii în calculator. Cum se poate realiza acest lucru? Foarte simplu: în loc să separe iremediabil DATELE de COD, programarea orientată pe obiecte nu face altceva decît să contopească cele două entităţi. Apropiindu-ne acum de C++, putem spune că acesta îmbină calităţile limbajului C cu avantajele programării obiectuale. O foarte bună caracterizare a rezultatului obţinut o face chiar Bjarne Stroustrup, omul ce a pus bazele noului limbaj de programare general, conceput astfel încît să-i facă pe programatorii serioşi să programeze într-o manieră cît mai plăcută.

Prezenta lucrare se adresează în primul rînd cunoscătorilor limbajului C dornici să-şi modernizeze stilul şi concepţia abordării proiectelor informatice. Avînd în vedere faptul că limbajul C++ este un superset al C-ului, pe întreg parcursul lucrării se vor presupune cunoscute noţiunile ce stau la baza programării în C. O parcurgere exhaustivă în cadrul unei singure cărţi a ambelor limbaje ar avea două neajunsuri: pe de o parte expunerea ar fi prea lungă (vezi cele 4 volume TURBO C++ scoase de BORLAND), iar pe de alta, informaţia specifică C++ s-ar dilua şi răspîndi nepermis de mult.

Începătorilor le recomandăm această carte, cu menţiunea că, înainte de parcurgerea ei, este necesară studierea unei cărţi de C clasic.

O notă specială: avînd în vedere că limba oficială a informaticii este engleza şi că - cel puţin pe moment - marea majoritate a termenilor de specialitate nu au o traducere scurtă şi satisfăcătoare în limba română (Ex: MOUSE, SCOPE, BINDING) pe parcursul lucrării vor fi utilizaţi şi termenii englezeşti.

Lucrarea de faţă descrie pe larg cel mai răspîndit dialect al limbajului C++ în lumea PC-urilor: TURBO C++.

Doresc să le mulţumesc pe această cale prietenilor mei Adrian Frits, Călin Caşcaval, Marius Greere şi Horaţiu Guja pentru observaţiile şi sfaturile

preţioase pe care mi le-au dat. Fără ajutorul lor, această carte nu ar fi existat în forma ei actuală.

AUTORUL

1. INTRODUCERE ÎN O.O.P.

1.1. Scopul O.O.P.

Termenul de OOP provine din englezescul "Object Oriented Programming" , care desemnează disciplina programării obiectuale (sau orientată-obiect). Domeniul OOP este deosebit de larg, şi are la bază ideea generoasă de a unifica două concepte de bază: datele aplicaţiei şi codul necesar tratării lor. Pentru a se ajunge la acest deziderat, se crează utilizatorului facilităţi deosebite în direcţia definirii unor tipuri de date proprii şi a unor operatori destinaţi manipulării lor. Acestea se vor comporta aidoma tipurilor de date şi operatorilor standard (întregi, numere reale, caractere, respectiv adunare, scădere ş.a.m.d.). Astfel, după necesităţi, un programator poate crea tipul COMPLEX sau MATRICE, care să opereze cu numere complexe sau matrici. O secvenţă banală în cazul numerelor reale, de genul

```
complex a,b,c;
matrice m,n,o;
...
c=a+b;
m=n*o;
```

poate deveni o realitate, chiar dacă cele şase variabile reprezintă numere complexe şi matrici. Noile tipuri de date ("complex" şi "matrice") se numesc CLASE , variabilele "a" , "b" , "c" , "m" , "n" şi "o" poartă numele de OBIECTE (sau instanţieri ale claselor "complex" şi "matrice"), iar operatorii redefiniţi "+" şi "*" se numesc METODE ale claselor în cauză. Prin unificarea datelor cu codul se înţelege că o CLASĂ conţine atît structurile de date necesare descrierii unui OBIECT, cît şi METODELE (funcţiile) cu ajutorul cărora obiectele în cauză vor fi manipulate.

Este evident că, în acest mod, gradul de abstractizare e mult mai ridicat, iar programele devin mai uşor de înţeles, depanat şi întreţinut.

Datorită modului de abordare fundamental diferit, nu trebuie să vă sperie faptul că, la început, veţi avea unele dificultăţi de adaptare. Cei care au făcut trecerea de la FORTRAN la PASCAL cunosc acea inerentă senzaţie de nesiguranţă la primele contacte cu recursivitatea şi backtracking-ul. În scurt timp, veţi remarca faptul că munca de pînă acum - în sensul de adaptare a unei probleme la modul de lucru al calculatorului - va fi facilitată de marea flexibilitate şi maleabilitate a abordării obiectuale.

1.2. Scurt istoric

În primele limbaje de programare evoluate, cea mai mare parte a atenţiei era acordată descrierii structurilor de date deoarece, practic, prelucrările ce se făceau asupra lor erau destul de simple şi nu necesitau un efort deosebit. Spre exemplu, putem cita cazul COBOL-ului (şi azi foarte răspîndit), în care circa 70% din lungimea codului sursă este ocupată de o descriere a structurilor de date. În timp, odată cu creşterea complexităţii programelor, a apărut necesitatea unei "ordonări" riguroase a muncii. Ca o treaptă superioară, capabilă să rezolve în bună măsură problemele ivite, s-a impus PROGRAMAREA STRUCTURATĂ. De-a lungul anilor, în special datorită creşterii dimensiunii produselor software, şi acest instrument a devenit la rîndu-i depăşit. Sporirea complexităţii programelor aducea după sine dificultăţi reale legate de rigiditatea şi întreţinerea unor programe mamut. Cu alte cuvinte, deşi preţul produselor era în creştere, calitatea lor tindea să scadă.

În căutarea unei soluţii care să scoată informatica din situaţia dificilă în care se afla, s-a pornit de la ideea că "o bună reprezentare a temei de rezolvat este deseori cauza transformării unei problemei complexe într-una deosebit de simplă" (P.H. WILSON, director al laboratorului de inteligenţă artificială de la M.I.T.). În urma diversificării căutărilor, au început să apară limbaje destinate soluţionării problemelor specifice anumitor domenii de activitate. Dintre tendinţele novatore merită să amintim cel puţin două: limbajele avînd la bază noţiunea de "CADRU" ("FRAME") şi cele ce pornesc de la ideea de "ACŢIUNE". În ciuda faptului că ambele sînt destinate inteligenţei artificiale, există totuşi o deosebire ce merită reţinută. Primele implementează OPERAŢII asupra unor MODELE DE ENTITĂŢI, pe cînd celelalte postulează faptul că OBIECTELE nu sînt simple elemente pasive asupra cărora se fac anumite prelucrări, ci, dimpotrivă, menirea OBIECTELOR rezidă în a ACTIVA prelucrările ce le vor suporta ele însele.

În fine, pornind de la noţiunea de CLASĂ, o altă tendinţă a dat naştere aşa-ziselor LIMBAJE ORIENTATE PE OBIECTE. Primele dintre ele cu adevărat demne de acest nume au fost SIMULA (1965) şi SIMULA-2 (1967). În anii '70 a apărut şi celebrul SMALLTALK. Cel mai mare dezavantaj al lor (din punct de vedere al penetrării pieţei) a fost, din cîte se pare, însuşi faptul că au apărut ca limbaje de sine stătătoare, avînd o răspîndire relativ redusă. Din acest motiv, puţini programatori erau dispuşi să abandoneze limbajele consacrate în acele momente, doar de dragul de a lucra obiectual.

Şi, cum diferenţele între punctul de vedere al cercetătorilor şi cel al marilor industriaşi au la bază însăşi deosebirea dintre ARTĂ şi RENTABILITATE, o bună bucată de vreme moda programării obiectuale a fost uitată. În anii 80, în urma acceptării definitive a limbajului C, un colectiv

condus de Bjarne Stroustrup încearcă introducerea conceptului de CLASĂ într-un dialect numit "C with Classes". Ideea prinde contur şi, în 1983, ia naştere prima versiune a noului limbaj: C++. Aproape imediat apar şi partizanii fanatici ai OOP-ului, aflat acum la o a doua tinereţe. Profitînd deci de marea sa popularitate în rîndul softiştilor şi de multitudinea domeniilor de aplicaţie (de la grafică interactivă la proiectarea interfeţelor utilizator şi de la exploatarea reţelelor de calculatoare la tehnicile de proiectare a compilatoarelor) "moftul" devine modă, iar moda certitudine.

De ce acest succes extraordinar? Mai ales din cauza faptului că limbajul C++ nu face nimic altceva decît să-i dea un nou avînt unuia dintre cele mai la modă limbaje ale momentului (este vorba de "C"). Cînd ne gîndim că toţi cei implicaţi în lumea UNIX-ului, şi nu sînt puţini, pot trece direct la OOP profitînd de păstrarea -practic neschimbată- a sintaxei şi regulilor de construcţie, lucrurile încep să se mai lămurească.

Cu toate acestea, problemele nu lipsesc. O declaraţie foarte sugestivă ar fi cea a "inventatorului" limbajului C++: <Utilizatorii au "sărit" pe C++ înainte ca specialiştii să aibă timpul necesar să-i instruiască de aşa manieră încît să-l poată folosi cu randament maxim!>. Ce ascunde această mărturisire? În principal două aspecte:

a. Lipsa timpului fizic a făcut ca formarea cadrelor de specialitate să nu urmeze ciclul optim. Acesta, se presupune că ar consta în a începe învăţarea programării cu un limbaj PUR obiectual, după care să se facă trecerea la programarea structurată. În acest mod, ambele tehnici ar putea fi perfect stăpînite. Vă veţi întreba care este dezavantajul? Răspunsul este destul de simplu. Dacă OOP-ul nu va fi bine înţeles, vor fi inevitabile situaţiile în care, de exemplu, un programator ce nu a învăţat corect PROLOG-ul este tentat tot timpul să simuleze în acest limbaj procedeele de bază învăţate în C. Sau, analog, să utilizeze şi în PASCAL schemele de simulare a recursivităţii deprinse în FORTRAN.
b. Ca o consecinţă directă a punctului "a.", în numărul din octombrie 1991 al revistei "INFO PC" este semnalat faptul că peste 60% din compilatoarele de C++ existente pe piaţă nu sînt folosite decît pentru dezvoltarea de software structurat, şi nu de tip OOP (altfel zis, se lucrează în C pe un compilator C++).

Nu se pot face anticipări asupra modului în care vor evolua tendinţele în "înaltele sfere ale software-ului", dar, nu ar fi de mirare, că, în cîţiva ani, larga popularitate a limbajului "C++" să nu ducă la o reîntoarcere la punctul de plecare: SMALTALK sau, de ce nu, SIMULA.

Şi ce altă confirmare a importanţei programării obiectuale ar fi mai semnificativă, decît faptul că broşurile de angajare (în domeniul software) la IBM sau MicroSoft cer tuturor viitorilor candidaţi cunoaşterea limbajului

1.3. Avantaje OOP

Vom începe prin a defini principalele concepte ce stau la baza OOP, după care ne vom putea da seama mai bine de avantajele lor.

Prin noul concept de ÎNCAPSULARE se ating mai multe obiective: "contopirea" datelor cu codul (în cadrul aşa numitelor CLASE), facilitatea deosebită de localizare a erorilor (întotdeauna cauza se află în "interiorul" unei singure clase) şi de modularizare a problemei de rezolvat (fiecare clasă va rezolva o anumită problemă).

Un alt termen introdus de OOP, MOŞTENIREA, aduce încă un avantaj specific. În urma definirii unei clase, cu un minim de efort şi timp, se pot preciza seturi de clase asemănătoare, avînd totuşi o trăsătură distinctivă. Un exemplu foarte sugestiv ar fi o clasă a mamiferelor, din care fac parte clasa cîinilor şi cea a primatelor. Cîinii se împart la rîndul lor în cockeri, boxeri, dogi, fox-terieri ş.a.m.d. Practic, cîinii şi primatele au atît unele trăsături comune (moştenite ca mamifere), cît şi altele specifice. Pe de altă parte, dogii şi cockerii au trăsături comune atît în calitate de cîini, cît şi ca mamifere. Dacă delimităm corect clasa mamiferelor (nasc pui vii, au sînge cald, îşi alăptează puii), în momentul în care definim cîinii, e suficient să spunem că sînt mamifere pentru ca toate caracteristicile de mai sus să fie subînţelese. Rămîn prin urmare de specificat acele trăsături ce le deosebesc de celelalte mamifere (carnivore, animale domestice, etc.). Întocmai se petrec lucrurile şi în universul claselor C++. În paginile cărţii veţi găsi dovada clară a facilităţilor aduse de către MOŞTENIRE: posibilitatea de extindere a unui program printr-un minim de efort.

La actualul nivel de cunoştinţe vom defini termenul de POLIMORFISM doar intuitiv, urmînd ca la momentul potrivit să dăm toate amănuntele necesare. Ideea de bază constă în faptul că, într-o ierarhie de clase obţinute prin moştenire, o metodă poate avea forme diferite de la un nivel la altul (specifice respectivului nivel din ierarhie). Pentru a lămuri cît mai bine această, problemă vom lua un exemplu concludent tot din lumea animală. Cosiderăm lumea vie drept o clasă generală avînd printre alte caracteristici şi funcţia (în limbaj OOP am spune METODĂ) de HRĂNIRE. Indiferent că e vorba de o lalea, un parameci, o balenă sau un jaguar, toate aceste vietăţi posedă funcţia de HRĂNIRE. Atîta doar că fiecare dintre ele o realizează altfel. Din acest punct de vedere, HRĂNIREA este un proces abstract, dar care în momentul asocierii lui cu o fiinţă vie devine concret. Niciodată cînd spunem că ne hrănim cîinele sau pisica nu ne gîndim să-i expunem la soare şi să-i stropim, ca pe două lalele.

Există şi alte avantaje ce ţin de mecanismul claselor, dar sînt destul de greu de explicat acum. Ele vor fi prezentate pe parcursul cărţii, în cadrul capitolelor corespunzătoare.

Vom încheia această secţiune prin a sublinia avantajele de ordin practic ale programării obiectuale.

În momentul de faţă, piaţa informatică este invadată de adevărate biblioteci şi colecţii de CLASE. Menirea lor este de a permite un coeficient cît mai ridicat de reutilizare a codului scris. Pe de altă parte, pornind de la aceste clase, utilizatorul poate crea prin moştenire alte clase, care să-l ajute să-şi rezolve problema în mod optim. În plus, programatorul are garanţia folosirii unor proceduri scurte, inteligibile şi, nu în ultimul rînd, CORECTE.

Un alt domeniu de utilizare a bibliotecilor de clase este cel al realizării PROTOTIPURILOR unor produse software. Prin PROTOTIP se înţelege un program funcţional, simplificat şi care să dea o imagine clară a produsului final. Utilitatea lor constă în faptul că, într-un timp foarte scurt şi cu un efort minim din partea producătorului, clientul poate avea o imagine destul de clară a produsului final. De ce am afirmat că bibliotecile de clase pot fi utile în realizarea de protitipuri? Pentru că, dacă în cadrul unei aplicaţii oarecare, avem nevoie de un algoritm de recunoaşteri de forme (de exemplu), nu este necesar ca prototipul să conţină un algoritm optim. Este preferabilă utilizarea unei metode deja implementate în cadrul unei biblioteci, şi care să facă acelaşi lucru. În acest mod, prototipul poate fi prezentat într-un interval de timp foarte scurt, după care se vor înlocui doar acele "module" care sînt deficitare din anumite puncte de vedere (consum excesiv de memorie, viteză nesatisfăcătoare, ş.a.m.d.).

Nu trebuie să neglijăm nici facilitarea de către OOP a cooperării între diversele echipe şi/sau persoane lucrînd la unul şi acelaşi proiect.

1.4. Înainte de început ...

... trebuie făcută o precizare: cartea de faţă nu va aborda TURBO C++ din punct de vedere al mediului de programare (IDE, modele de memorie, diverse utilitare, ...), ci doar ca dialect C++. Toate programele prezentate în carte au fost testate cu ajutorul unui compilator TURBO C++ versiunea 2.00, sub sistemul de operare MS-DOS 5.0.

Şi încă trei observaţii:

a. Implicit, orice fişier care conţine cod "C++" va trebui să aibă extensia ".CPP" pentru a fi compilat corect.
b. Toate programele ce utilizează biblioteca grafică BORLAND conţin un

apel de funcţie de genul

```
initgraph(gd,gm,"c:\\tcpp\\bgi");
```

Pentru ca aceste exemple să poată fi executate corect pe calculatorul dumneavoastră, va trebui să aveţi grijă să modificaţi cel de al treilea parametru de aşa manieră încît să exprime calea corectă spre directorul ce conţine fişierele specifice modului de lucru grafic.

2. SCURTĂ INCURSIUNE ÎN C++

Scopul acestui capitol este de a face o trecere în revistă a celor mai importante caracteristici şi concepte introduse de limbajul C++. Nu se doreşte o discuţie exhaustivă, ci doar o vedere de ansamblu asupra a ceea ce se va discuta în amănunt în următoarele capitole. Prin cîteva exemple destul de simple se va iniţia un prim contact al cititorului cu noile concepte. Explicaţiile furnizate în cadrul acestui capitol vor fi de natură să incite la o parcurgere atentă a celorlalte secţiuni, în care gradul de rafinare al informaţiei va creşte substanţial.

Practic, din cuprinsul acestui capitol nu sînt "obligatoriu" de citit decit primele patru secţiuni. Celelalte conţin doar informaţii ce vor fi prezentate mai pe larg în cadrul capitolelor următoare. Ideea de a scrie un capitol "jumătate facultativ" vine în întîmpinarea celor mai cunoscute două tehnici de învăţare: "BOTTOM-UP" (se începe de la zero, şi se va clădi încet, dar sigur, un edificiu ale cărui dimensiuni vor fi percepute doar în finalul activităţii de învăţare) şi "TOP-DOWN" (se începe printr-un prim contact cu problema în toată complexitatea ei, după care se va trece la aprofundarea diverselor componente). Adepţii primului stil vor putea sări liniştiţi peste a doua jumătate a capitolului. Cu toate că oferim posibilitatea alegerii, noi recomandăm cea de a doua abordare, care are avantajul de a permite o corectă încadrare a fiecărui amănunt în contextul general.

2.1. Intrări şi ieşiri. Stream-uri

Aproape toate cărţile de C dau drept prim exemplu de program celebrul "Hello world !", care afişează acest text pe ecran folosind funcţia "printf". O primă dovadă de non-conformism a limbajului C++ este însuşi faptul că reuşeşte cu mult succes înlocuirea clasicelor funcţii "printf" şi "scanf" (cîţi începători nu au avut dificultăţi cu argumentele - pointeri - ai acesteia din urmă!) cu o bibliotecă ce are la bază conceptul de STREAM. Avînd în vedere că acest concept fundamental este descris pe larg în capitolul 6, să vedem doar cum se operează cu el:

```
# include <iostream.h> // header-ul bibliotecii de stream-uri

void main(void){
 int a;
/* declaratia de mai jos "float b;" va fi comentata conform celeilalte
posibilitati C++ de a scrie o linie de comentariu */
 float b;// aceasta este cea de a doua maniera de comentariu C++
```

```
char c;

cout<<"\n\nTastati un intreg : ";
cin>>a;
cout<<"\nTastati un numar real : ";
cin>>b;
cout<<"\nTastati o litera : ";
cin>>c;
cout<<"\n\nAti tastat "<<a<<" , "<<b<<" , "
   <<c<<".\n";
}
```

În urma execuţiei programului pe ecran se va afişa:

Tastati un intreg : 12

Tastati un numar real : 12.34

Tastati o litera : a

Ati tastat 12 , 12.34 , a.

Aşa cum specifică şi comentariul, "iostream.h" este header-ul unei biblioteci de stream-uri de intrare şi ieşire. El vine să înlocuiască pe "stream.h", header-ul bibliotecii de stream-uri corespunzătoare primei versiuni a limbajului C++. O altă observaţie referitoare la aceeaşi primă linie ar fi legată de noua posibilitate de comentariu, şi anume cea utilizînd două slash-uri. Cele două caractere "/" semnalizează începutul comentariului, el continuînd pînă la sfîrşitul liniei respective.

Probabil aţi intuit deja că "cin" şi "cout" desemnează dispozitivele de intrare şi ieşire (implicit tastatura şi ecranul). Operatorii "<<" şi ">>", folosiţi în C pentru shift-ări, sînt utilizaţi de astă dată drept "pune la ..." şi "preia de la ...". După cum vom vedea mai tîrziu, limbajul C++ permite o astfel de redefinire a operatorilor existenţi. Chiar dacă un operator este redefinit, se vor păstra şi vechile lui sensuri. Atît că, în funcţie de operanzii care-l apelează, se va efectua o operaţie sau alta.

2.2. Funcţii cu parametrii impliciţi. Funcţii cu acelaşi nume

Începem această secţiune prin a relua o afirmaţie făcută în cuvîntul introductiv. DEOARECE LIMBAJUL C++ ESTE UN SUPER-SET AL

LIMBAJULUI C, TOATE DECLARAŢIILE ŞI DEFINIŢIILE DE VARIABILE ŞI FUNCŢII, CARACTERISTICE DIALECTULUI TURBO C, VOR RĂMÎNE ÎN CONTINUARE VALABILE. Cum ne-am propus doar prezentarea caracteristicilor C++, nu vom explica în cele ce urmează nici un fel de instrucţiuni care, asemeni definirii funcţiei "exemplu", nu necesită decît cunoaşterea limbajului C:

```
int exemplu(int p)
{
 int i;
     for(i=0;i<p;i++)
     printf("\n\t i=%d",i);
     return p/2;
}
```

Vom continua prin a analiza cu atenţie exemplul de mai jos:

```
# include <iostream.h>

void functie(int a=123,double b=456.789,
              char * c="program")

{
 cout<"\na="<<a<<" b="<<b<<" c="<<c;
}

void main(void)
{
 functie(321,987.654,"subrutina");
 functie(321,987.654);
 functie(321);
 functie();
}
```

În urma execuţiei secvenţei de instrucţiuni, pe ecran vor apărea următoarele şiruri de caractere:

a=321 b=987.654 c=subrutina
a=321 b=987.654 c=program
a= 321 b= 456.789 c= program
a=123 b=456.789 c=program

Probabil că intuiţi deja despre ce este vorba: prin declaraţia

```
void functie(int a=123,double b=456.789,
              char * c="program");
```

compilatorul este atenţionat că orice apel al funcţiei "functie" poate avea între

zero şi trei parametri actuali. În caz că numărul lor este mai mic de trei, cei lipsă vor lua IMPLICIT valorile specificate în lista de parametri formali. Dar atenţie! Această facilitate are şi ea limitele ei: indiferent de numărul parametrilor actuali ai funcţiei, ordinea şi tipul lor sînt întotdeauna aceleaşi! Ba chiar mai mult, începînd cu primul parametru transmis implicit, TOŢI cei care-i urmează în listă nu mai pot fi transmişi DECÎT cu valori implicite!

"Functions overloading" - în traducere liberă "suprapunerea" ori "supraîncărcarea" funcţiilor - este un alt concept fundamental în C++. Graţie lui, utilizatorul poate să declare funcţii cu acelaşi nume, dar avînd prototipuri diferite (altfel spus, este vorba de funcţii cu acelaşi nume, dar cu semnături diferite). Aceste funcţii pot fi distinse prin faptul că vor diferi fie prin tipurile parametrilor formali, fie prin tipul valorii returnate. Vom începe prin a da un exemplu, pe care-l vom comenta în continuare.

```
# include <iostream.h>
# include <string.h>

// in TURBO C++ declaratia ce urmeaza nu este obligatorie
 overload a;

// prototipurile celor doua functii avind acelasi nume

void a(void);
int a(char *);

// definitiile celor doua functii

void a(void){
     cout<<"\n Nu exista parametri !\n";
 }

 int a(char *s){
    cout<<"\n Parametrul este pointer la
          stringul : "<<s;
    return strlen(s);
 }

void main(void){
 a();
 cout<<"\n Lungimea stringului = "<<a("asdfe");
}
```

Pe ecran se vor afişa următoarele texte:


```
Nu exista parametri !

Parametrul este pointer la stringul : asdfe
Lungimea stringului = 5
```

La versiunea iniţială (AT&T 1.0) a lui Bjarne Strostroup, în C++ era necesară utilizarea cuvîntului cheie "overload", urmat de numele funcţiei respective. În noile implementări, acest lucru nu mai este impus. Totuşi, recomandăm folosirea unei astfel de declaraţii, în ideea că un program trebuie scris cît mai explicit. Pentru că, nu uitaţi, programele trebuie să fie uşor de înţeles atît pentru cel care le-a făcut, cît şi pentru o altă persoană ce ar urma să-i continue munca!

În situaţia de faţă, utilitatea unui astfel de exemplu poate fi îndoielnică. Dacă însă dorim lărgirea bibliotecii matematice a limbajului C++ cu noi funcţii (de exemplu ridicarea la putere), poate prezenta un cert interes existenţa unor funcţii cu acelaşi nume ("PUTERE"), dar care operează cu diverşi parametrii (numere complexe sau raţionale).

Exemplu:

```
typedef struct rational1{
 float numarator,numitor;
 } rational1;

typedef struct complex1 {
 float p_reala , p_imaginara;
 } complex1;

overload PUTERE;
rational1 * PUTERE(rational1 * , int );
complex1 * PUTERE(complex1  *,int );
```

2.3. Referinţa

În absolut toate domeniile informaticii se pune foarte acut problema eficienţei. Ea comportă două aspecte: unul legat de spaţiul de memorie utilizat şi un altul legat de timpul de execuţie al programelor. În legătură cu acesta din urmă, este unanim cunoscută preferinţa pentru limbajul de asamblare a tuturor programatorilor interesaţi de economisirea fiecărei fracţiuni de secundă. Dar, de la acest aspect pînă la dimensiunile pe care le ia astăzi fenomenul, este o cale foarte lungă. Vom da două exemple elocvente în acest sens.

Specialiştii de la MicroSoft au tras concluzia că folosirea specificatorului

"pascal" în faţa funcţiilor scrise în C scurtează codul generat cu 7%. Acest cîştig se obţine prin faptul că, în convenţia PASCAL, stiva este gestionată de către rutina apelată, nu de către cea apelantă.

Al doilea exemplu este rezultatul studiilor lui Bjarne Strostroup. Acesta afirmă că timpul consumat de un apel de funcţie de tipul "alloc" este atît de mare încît este mai rentabil ca, în cazul unor alocări repetate de memorie pentru date de dimensiuni reduse, să se proiecteze un "gestionar" al memoriei. Acesta va aloca la un moment dat o cantitate mai mare de memorie, din care să "livreze", la comandă, cantităţile necesare.

Revenind, să amintim că, în cazul unor apeluri de funcţii care au drept parametri obiecte de mari dimensiuni, este foarte necconomică punerea acestora pe stivă la apelul funcţiei. O economie semnificativă se poate obţine prin utilizarea pointerilor. Aceasta are, însă, două dezavantaje:

a. este destul de puţin elegantă deoarece, în unele cazuri, este mult mai sugestivă operarea asupra obiectelor decît cea asupra pointerilor la ele;
b. există cazuri speciale în C++ cînd o astfel de soluţie este contraindicată.

De aceea s-a făcut apel la un nou tip de dată, numit tipul REFERINŢĂ. Acesta implementează perfect conceptul de "transmitere a parametrilor prin referinţă". O referinţă reprezintă, de fapt, numele unui obiect. Dacă avem tipul de dată "T", prin "T&" se va înţelege o referinţă la un obiect de tipul "T". Orice referinţă trebuie iniţializată, după care devine pur şi simplu un alt nume (un "alias") al obiectului cu care a fost iniţializată.

Pentru a exemplifica cît mai bine eficienţa utilizării referinţelor, vom analiza două exemple: unul care foloseşte şi unul care nu foloseşte tipul referinţă. Programul de mai jos nu va folosi tipul referinţă, pe cînd cel care-i urmează îl va utiliza:

```
# include <iostream.h>
# define MAX_NUME 25

 typedef struct _persoana {
    char nume[MAX_NUME],prenume_1[MAX_NUME],
           prenume_2[MAX_NUME],initiala_tatalui;
    int virsta,cod;
    float inaltime;
 } persoana;

void tipareste_persoana(persoana p){
 cout<<p.nume<<" "<<p.prenume_1<<" "<<p.prenume_2;
 cout<<"    virsta : "<<p.virsta<<" , cod ="<<
     p.cod<<" , inaltimea = "<<p.inaltime<<"\n\n";
```



```
}

void main(void){
 persoana James_Bond={"Bond","James","",'K',40,
                          007,1.88};
 tipareste_persoana(James_Bond);
 persoana Ion_Ion=James_Bond;
 tipareste_persoana(Ion_Ion);
}
```

Pe ecran se va tipări:

Bond James virsta : 40 , cod = 7 , inaltimea = 1.88

Bond James virsta : 40 , cod = 7 , inaltimea = 1.88

Pentru înţelegerea programului, trebuie subliniat că instrucţiunile de genul

```
persoana Ion_Ion=James_Bond;
```

realizează iniţializarea printr-o copie bit cu bit a celei de a doua structuri în cadrul primeia.

Structura "persoana" ocupă un număr semnificativ de octeţi. Dacă se doreşte tipărirea unui şir de 1000 de persoane, timpul ocupat de transmiterea succesivă a celor 1000 de structuri ca parametri creşte nepermis de mult. Pentru a scăpa de acest inconvenient, vom folosi un al doilea program, care utilizează drept parametri referinţe la structuri de tip "persoana".

Iată acum cel de-al doilea program, în care se va utiliza tipul referinţă:

```
# include <iostream.h>
# define MAX_NUME 25

 typedef struct _persoana {
    char nume[MAX_NUME],prenume_1[MAX_NUME];
    char prenume_2[MAX_NUME],initiala_tatalui;
    int   virsta,cod;
    float inaltime;
 } persoana;
 persoana sir_persoane[20];

void tipareste_persoana(persoana& p){
 cout<<p.nume<<" "<<p.initiala_tatalui<<". "<<
    p.prenume_1<<" " <<p.prenume_2<<"\n";
 cout<<" virsta : "<<p.virsta<<" , cod = "<<
    p.cod<<" , inaltimea = "<<p.inaltime<<"\n\n\n";
```

```
}

void actualizeaza_virsta(persoana& refPersoana)
{
 refPersoana.virsta++;
}

persoana& cauta_persoana_de_50_ani(void)
{
 for(int i=0;i<20;i++)
    if(sir_persoane[i].virsta==50)
         return sir_persoane[i];
}

void main(void)
{
 cout<<"\n\n\n\n\nProgram demonstrativ :
   REFERINTE\n\n\n";
 persoana James_Bond={"Bond","James","",'K',40,
                      007,1.88};

 persoana& ref_pers1=James_Bond;
 tipareste_persoana(ref_pers1);

 persoana Ion_Ion=James_Bond;
 persoana& ref_pers2=Ion_Ion;
 tipareste_persoana(ref_pers2);

 actualizeaza_virsta(ref_pers2);
 tipareste_persoana(ref_pers2);

 tipareste_persoana(James_Bond);

 persoana aa={"Iftode","Ilie","",'K',40,007,1.88};

 for(int i=0;i<20;i++)
 {
    // initializare referinta
    persoana& ref_pers3=sir_persoane[i];
    sir_persoane[i]=aa; // initializare element din sir
    for(int j=-1;j<i;j++)
       actualizeaza_virsta(ref_pers3);
 }
```



```
 cout<<"\n\nPersoana de 50 de ani : \n";
 tipareste_persoana(cauta_persoana_de_50_ani());
}
```

În urma execuţiei programului pe ecran se va afişa:

Program demonstrativ: REFERINTE
Bond K. James
virsta : 40 , cod = 7 , inaltimea = 1.88

Bond K. James
virsta : 40 , cod = 7 , inaltimea = 1.88

Bond K. James
virsta : 41 , cod = 7 , inaltimea = 1.88

Bond K. James
virsta : 40 , cod = 7 , inaltimea = 1.88

Persoana de 50 de ani :
Iftode K. Ilie
virsta : 50 , cod = 7 , inaltimea = 1.88

De astă dată, în funcţia "tipareste_persoana" nu va fi transmisă întreaga structură, ci doar adresa ei. Parametrul "refPersoana" va referi structura de tipul "persoana" care apărea pe post de parametru formal. De remarcat că membrii structurii se referă prin ".", nu prin "->" (să nu uităm că o referinţă la un obiect nu este altceva decît un ALT NUME desemnînd exact aceeaşi instanţiere cu care a fost iniţializată).

După cum se va vedea în urma rulării programului, acţiunile ce se vor executa în "actualizeaza_virsta" vor fi păstrate în cadrul obiectelor şi în urma părăsirii acestei funcţii. Acesta este un al doilea aspect al utilizării referinţelor: se lucrează direct asupra obiectului referit. De remarcat că şi în cazul utilizării referinţelor, iniţializările de obiecte prin atribuire au loc exact în maniera descrisă mai sus (adică tot prin copiere bit cu bit).

În cadrul ultimelor două proceduri se remarcă un lucru foarte interesant. Restricţia din limbajul C, legată de obligativitatea declarării tuturor variabilelor locale la începutul procedurii, nu mai e valabilă (vezi locul în care sînt declarate "aa" şi "ref_pers3"). Mai mult chiar, se vede că, în două rînduri, contorul întreg "i" este definit în cadrul instrucţiunii "for". Totuşi se mai cuvine făcută o observaţie: instrucţiunile de genul

```
for(int i=0;.....)
```

sînt echivalente cu

```
int i;
for(i=0;...)
```

Altfel spus, o secvenţă

```
for(int i=0;...)
{
   // ....
}
for(int i=0;...)
```

va provoca eroarea de "redefinirea variabilei i". Am atras atenţia asupra acestui fapt deoarece multă lume este tentată să considere că declararea lui "i" în cadrul unui "for" îl face să devină o variabilă locală acestuia. Deci nu uitaţi: domeniul de existenţă al lui "i" ia sfîrşit la închiderea blocului ce-l conţine pe "for".

2.4. NEW şi DELETE

Programele scrise în TURBO C neobiectual foloseau pentru alocarea dinamică de memorie familiile de funcţii "free" şi "malloc", care erau destul de greoaie în folosire. Ele au fost păstrate în TURBO C++ doar pentru menţinerea compatibilităţii cu versiunile precedente. În locul lor, se pot folosi cei doi operatori creaţi în acest scop. Primul, "new", serveşte la alocarea dinamică a memoriei, iar cel de al doilea - "delete" - la eliberarea ei.

Operatorul "new" va returna un pointer la zona de memorie alocată dinamic. În cazul în care nu există suficientă memorie disponibilă, alocarea nu va avea loc. Acest fapt este semnalizat prin returnarea unui pointer NULL. De aceea, se recomandă că, după fiecare utilizare a lui "new", să se testeze valoarea returnată. În caz de eşec al operaţiei de alocare, este bine să se tipărească un mesaj de avertizare şi să se încheie execuţia programului. Altfel există riscul de a se obţine efecte dezastruoase.

Operatorul "delete" va elibera zona de memorie la care pointează argumentul său.

În programul de mai jos se poate vedea un exemplu de testare a corectitudinii alocării dinamice de memorie.

Exemplu:

```
# include <iostream.h>
# include <process.h>
```



```
# define nl "\n"

 typedef struct _pereche {
    int a,b;
    struct _pereche *next;
 } pereche;

// Testeaza valoarea parametrului actual. Daca el este NULL
// inseamna ca ultima operatie de alocare dinamica de memorie
// s-a soldat cu un esec. Din acest motiv se afiseaza un mesaj
// de avertizare si se provoaca oprirea executiei programului.
void testare_pointer(void * pointer){
 if(pointer==NULL)
 {
    cout<<"\n\nMemorie insuficienta !!!\n";
    exit(1);
 }
}

void main(){
 pereche *p1,*p2;

 p1=new pereche;   // se aloca memorie pentru o structura
 testare_pointer((void *) p1); // test alocare corecta
 p2=new pereche[10]; // se aloca memorie pentru un
                     // sir cu 10 elemente
 testare_pointer((void *) p2); // test alocare corecta
 p2[2].a=10;
 cout<<nl<<p2[2].a<<nl;
 delete p1; // se elibereaza memoria alocata pentru o structura
            // ...
}
```

O mare parte din exemplele cărţii vor utiliza operatorul "new". În cele mai multe din cazuri, nu s-a testat valoarea returnată de acesta. Omisiunea este INTENŢIONATĂ şi are două scopuri: pe de o parte se evită încărcarea excesivă a exemplelor, iar pe de alta nu se distrage atenţia de la scopul secvenţei de instrucţiuni.

2.5. Clase

În cadrul acestei secţiuni vom descrie succint una dintre cele mai impor-

tante noutăţi aduse de limbajul C++, şi anume conceptul de clasă. Cu ajutorul claselor se realizează un vechi deziderat al programatorilor: unificarea datelor şi a procedurilor.

Pentru a înţelege cît mai bine ideea ce stă la baza conceptului de clasă, vom analiza pentru început ideea de tip de dată, sub forma în care există ea în limbajul C. Spre exemplu, dacă ne referim la numerele întregi ("int"), vom remarca faptul că acestea sînt descrise de două trăsături: în primul rînd spaţiul de memorie în care numerele sînt reprezentate (şi care variază de la un calculator la altul), iar în al doilea rînd un set de operatori cu ajutorul cărora întregii pot forma expresii aritmetice sau logice. Programatorul nu este interesat de soluţia folosită de compilator pentru rezolvare a celor două aspecte, ci doar de modul în care poate utiliza respectivul tip de dată.

În ciuda facilităţilor de limbaj cu totul remarcabile, C-ul nu poate (şi nici nu are cum) să fie dotat cu toate tipurile de date şi operaţii pe care şi le-ar putea dori diverşii utilizatori. O rezolvare elegantă a acestui inconvenient este adusă de C++ prin conceptul de clasă. Cu ajutorul claselor, un programator îşi poate defini orice tip de dată dorit, împreună cu setul necesar de operaţii. Toate aceste informaţii vor fi reunite (încapsulate) într-o CLASĂ. Astfel, prin clasă vom înţelege un nou tip de date, definit de utilizator. Clasele pot fi folosite întocmai ca şi tipurile de date predefinite. Făcînd abstracţie de modul de realizare a celor spuse pînă acum, ne putem închipui foarte bine următoarea secvenţă de instrucţiuni C++:

```
complex a(10,1),b(2,3),c;
rational d(4,3);
c=a*b;
d=d*2;
```

În acest mod utilizatorul va opera cu numere complexe şi raţionale exact ca şi cum ar folosi tipurile predefinite "int"sau "float".

Înainte de a da un exemplu concret de implementare a unei clase, vom explica sumar modul de creare a unui PROJECT în mediul TURBO C++. Facem acest lucru deoarece la noul produs al firmei BORLAND o astfel de operaţie nu mai are loc manual, ci se efectuează cu sprijinul unor funcţii speciale ale meniului. Celor care nu cunosc tehnica de lucru cu PROJECT-uri le vom recomanda să studieze un manual de utilizare TURBO C. În cîteva cuvinte totuşi, despre ce este vorba. În cazul programelor de dimensiuni mai mari, scrierea codului în cadrul unui singur fişier e neplacută şi neeconomicoasă: de fiecare dată se vor compila toate liniile programului, chiar dacă s-a modificat un simplu comentariu. De aceea, se recomandă împărţirea programului în module mai scurte, care să fie scrise în fişiere separate. Aceste fişiere vor fi compilate independent, iar prin link-editarea lor în comun (în cadrul unui PROJECT) va lua naştere un singur program executabil. Avantajele sînt multiple: pe de o parte un astfel de produs software este mai uşor de gîndit şi întreţinut, iar pe de alta, în cadrul acţiunii de modificare a unuia dintre module, nu va fi recompilat decît modulul în cauză.

Revenind la mecanismul de creare/actualizare a PROJECT-urilor TURBO C++, acesta este implementat în cadrul unor funcţii din meniul de bază. Astfel, în urma selectării din meniul principal a opţiunii PROJECT, se va putea fie crea, fie încărca un nou PROJECT (opţiunea OPEN PROJECT). În cadrul acestuia se vor putea folosi opţiunile INSERT şi DELETE, cu ajutorul cărora se adaugă/şterg fişiere la/din PROJECT-ul selectat. O dată creat un PROJECT, prin comanda RUN se vor compila şi link-edita toate fişierele sale, creîndu-se un singur program executabil.

Am făcut paranteza de mai sus deoarece în cadrul acestei secţiuni vom prezenta un prim exemplu scris modular, care va servi drept obiect de studiu atît pentru noţiunea de CLASĂ, cît şi pentru cea de PROJECT.

Începem cu un program compus din trei fişiere: STTNG_CL.DAT, SETTINGS.CPP şi MAIN_SET.CPP. Pentru a obţine din cele trei fişiere un program executabil este necesară crearea unui PROJECT (să-i zicem SETTINGS.PRJ), care să asigure link-editarea corectă a lui SETTINGS.CPP şi MAIN_SET.CPP (STTNG_CL.DAT va fi inclus ca header în celelalte două fişiere). Să vedem deci ce conţine fişierul STTNG_CL.DAT.

```
// DECLARAREA CLASEI "settings"
# include <graphics.h>

class settings {
   private:
   int cul,bkcul; // culoare de scriere si de fond
   struct textsettingstype txt; // atributele de scriere
                                // a textelor
   struct fillsettingstype fll; // atributele de umple-
                                // re a figurilor
   public:
   void put_settings(void);
   void get_settings(void);
};
```

Odată făcut acest lucru, să încercăm să accentuăm două definiţii. O CLASĂ este un tip abstract de date, definit de către programator. Un OBIECT este o variabilă declarată ca fiind de tipul CLASĂ. Cu alte cuvinte, un OBIECT este o INSTANŢIERE a unei CLASE.

Clasa pe care tocmai am declarat-o poartă numele "settings" (acesta este numele noului tip de dată pe care l-am definit). Ea are două secţiuni: cea

privată şi cea publică. În cazul nostru (foarte particular, de altfel) prima conţine doar declaraţii de variabile, iar a doua numai prototipuri de funcţii. Acest fapt se datorează unei pure întîmplări, în ambele secţiuni putînd figura, fără restricţii, atît variabile cît şi funcţii. Secţiunea privată păstrează (invizibil pentru utilizatorul clasei) informaţii folosite de către implementatorul clasei, pe cînd cea publică reprezintă interfaţa dintre clasă şi potenţialul ei utilizator. Pentru o mai bună înţelegere, vom face o analogie cu tipul de date "int" din C. În acest caz atît octeţii în care e memorat întregul, cît şi operatorii existenţi (+, -, * ş.a.m.d.) se află la dispoziţia utilizatorului, deci ele sînt publice.

O precizare care va lămuri multe neclarităţi: un "struct" din limbajul C clasic poate fi privit ca o clasă cu toate variabilele publice şi neavînd nici o funcţie membru. Am definit astfel şi funcţiile membru: acele funcţii ce sînt declarate în cadrul unei clase. Şi fiindcă tot a venit vorba de structuri, să spunem că, în C++, şi lor li se pot adăuga funcţii membru. În acest moment este clar că o secvenţă ca cea de mai jos nu mai reprezintă o enigmă pentru dumneavoastră.

```
class _clasa {
 // ...
 public : int i;
 void do_it(void) { cout<<i; };
 // ...
};

// ...

_clasa my_class; // declararea obiectului "my_class"
my_class.i=5;       // setarea membrului "i" la 5
my_class.do_it(); // apelarea functiei membru "do_it"
```

Să reluăm în cuvinte cele cîteva idei puse în evidenţă de acest scurt exemplu. Atît variabilele cît şi funcţiile membru ce apar în secţiunea publică a unei clase pot fi utilizate în orice punct al programului. Există totuşi o restricţie: ele pot fi accesate doar prefixate de către un nume de obiect (instanţiere a clasei respective). Operatorul ce face posibilă selectarea membrilor unui obiect este punctul ("."), întocmai ca în cazul structurilor din C. În schimb, toate variabilele şi metodele private nu vor putea fi accesate DECÎT de către alte metode ale clasei în cauză (trebuie să subliniem că în terminologia OOP "metoda" este sinonimă cu "funcţie membru").

În continuare, vom analiza fişierul "SETTINGS.CPP", care conţine descrierea funcţiilor membru ale clasei "settings".


```
/* Salveaza/restaureaza atributele modului de lucru grafic */
# include "sttng_cl.dat"  // e inclus ca header

// Restaureaza atributele modului de lucru grafic salvate cu
// get_settings()
void settings::put_settings(void)
{
 settextstyle(txt.font,txt.direction,txt.charsize);
 settextjustify(txt.horiz,txt.vert);
 setfillstyle(fll.pattern,fll.color);
 setcolor(cul);
 setbkcolor(bkcul);
}

// Salveaza atributele curente de lucru in mod grafic
void settings::get_settings(void)
{
 gettextsettings(&txt);
 getfillsettings(&fll);
 cul=getcolor();
 bkcul=getbkcolor();
}
```

Din cîte se vede, fişierul STTNG_CL.DAT este inclus sub formă de header. El a fost conceput ca fişier separat tocmai pentru a feri utilizatorul de munca plicticoasa de a tasta informatiile incluse in acesta ori de cite ori intr-unul dintre modulele unui PROJECT este utilizat un astfel de obiect. Pentru toţi cei care cunosc nucleul grafic al firmei BORLAND, scopul în care a fost creată clasa SETTINGS este evident. Ea realizează un mecanism simplu şi comod de salvare/restaurare a unor caracteristici specifice modului de lucru grafic (culoare de scriere/fond, font, direcţie de scriere, dimensiune caracter, şablon de umplere a figurilor etc.).

Vom mai sublinia că, la definirea funcţiilor membru, numele acestora este prefixat de numele clasei urmat de operatorul "::". Această construcţie indică faptul că este vorba de funcţia membru a unei clase anume. O astfel de măsură de prevedere este necesară deoarece în cadrul unui program putem avea mai multe clase care posedă funcţii membru cu acelaşi nume. De asemenea, poate exista o funcţie membru avînd acelaşi nume ca şi o funcţie nemembru.

În fine, vom prezenta fişierul MAIN_SET.CPP, care prezintă demonstrativ calităţile clasei SETTINGS.

```
# include <conio.h>
# include "sttng_cl.dat"//header ce descrie clasa SETTINGS;
                        //prezenta lui este necesara in caz
```

```
                                //ca se doreste folosirea de obiecte
                                //de acest tip
void main()
{
 int gd,gm;

 detectgraph(&gd,&gm);
 initgraph(&gd,&gm,"\\tcpp\\bgi");
 outtextxy(50,100,"HELLO !");

 getch();
 cleardevice();

 settings my_settings;
 my_settings.get_settings();

 settextstyle(GOTHIC_FONT,HORIZ_DIR,4);
 outtextxy(100,100,"HELLO !");
 settextstyle(SMALL_FONT,VERT_DIR,7);
 outtextxy(500,100,"HELLO !");

 my_settings.put_settings();
 getch();

 cleardevice();
 outtextxy(100,100,"HELLO !");

 getch();
 closegraph();
}
```

2.6. Constructori şi destructori

Să revenim la analogia între o clasă oarecare şi tipul de date "int". În cele expuse atunci, am precizat o serie de asemănări. De astă dată ne vom pune o problemă de altă natură: cine şi cînd alocă cei doi octeţi (în general acesta este numărul de bytes pe care se reprezintă un întreg) necesari memorării fiecărei variabile întregi? Răspunsul nu ne interesează din punct de vedere al corectitudinii, ci prin prisma sugestivităţii sale. De aceea ne putem închipui foarte bine că în momentul întîlnirii unei declaraţii de genul:

```
int i;
```


există o funcţie specială "int()" care rezervă numărul de octeţi necesari. În cazul în care se întîlneşte o formă de genul:

```
int j=9;
```

se va apela "int(9)", care, pe lîngă alocarea octeţilor necesari, îi mai şi iniţializează cu valoarea 9.

Un mecanism absolut similar trebuie să existe şi pentru eliberarea memoriei respective în momentul în care se părăseşte domeniul de existenţă al variabilelor. Dacă celor două funcţii le vom da numele generice de "constructor" şi respectiv "destructor", putem spune că am definit alte două concepte esenţiale ale programării obiectuale.

De remarcat că o clasă poate avea mai mulţi constructori. De exemplu, chiar în analogia noastră cu numerele întregi, avem de-a face cu doi constructori. Unul nu are nici un parametru (rezervă doar spaţiu), pe cînd al doilea are un parametru ce constituie valoarea iniţială a variabilei. De notat că, dacă o clasă este prevazută cu constructori, la fiecare declarare a unui obiect de acest tip va avea locapelul unuia dintre constructori. Tot acum trebuie subliniat că ÎNTOTDEAUNA constructorul va purta numele clasei în cauză. La fel şi destructorii, atît ca numele respectiv va fi precedat de un caracter " ~ ".

În continuare, vom da un exemplu de clasă avînd atît constructor, cît şi destructor.

```
// - clasa STACK implementeaza o stiva ale carei elemente sint
//   siruri de caractere

// - stiva este simulata printr-o lista simplu inlantuita

# include <iostream.h>
# include <process.h>
# include <string.h>
# include <stdio.h>

class stack {
 struct _stiva {
    char info[20];
    struct _stiva *next;
 } *head;      // pointer la inceputul stivei de stringuri

public:
 stack(void);    // CONSTRUCTOR !!!
 ~stack(void);   // DESTRUCTOR !!!
```

```
 void pop(void);
 void push(char *);
 void print_stack();
};

// Initializeaza lista ca fiind vida
inline stack::stack(void)
{
 head=NULL;
}

// Distruge stiva (lista simplu inlantuita)
stack::~stack(void)
{
 struct _stiva *a; // var.tampon
 if(head!=NULL)
 do
 {
     a=head->next;
     delete head;
     head=a;
 }
 while (a!=NULL);
}

// Tiparestestiva
void stack::print_stack(void)
{
 struct _stiva *a;     // var. tampon
 a= head;
 if(head!=NULL) {
     cout<<"\n STIVA CONTINE :\n\n";
     do {
         cout<<a->info<<"\n";
         a=a->next;
     }
     while (a!=NULL);
     cout<<"\n";
 }
 else cout<<"\n STIVA ESTE VIDA !";
}
```



```
// Descarca un element de pe stiva (elimina primul element din lista)
void stack::pop(void)
{
 struct _stiva *a;
 if (head!=NULL) {
     a=head;
     head=head->next;
     delete a;
 }
 else cout<<"Nu se poate executa \"pop\" dintr-o
          stiva goala !";
}

// Pune un element pe stiva (adauga la inceputul listei un nou
// element)
void stack::push(char *info)
{
 struct _stiva *a=new struct _stiva;
 if (a==NULL)
 {
     cout<<"Memorie insuficienta in STACK::PUSH\n";
     exit(1);
 }
 strcpy(a->info,info);
 a->next=head;
 head=a;
}

void main(void)
{
 stack my_stack;

 my_stack.push("Aceasta");
 my_stack.push("este");
 my_stack.push("stiva");

 my_stack.print_stack();
 my_stack.pop();
 my_stack.print_stack();

 my_stack.pop();
 my_stack.pop();
 my_stack.pop();
```

```
 my_stack.print_stack();
 my_stack.push("Sirsit");
 my_stack.print_stack();
}
```

În urma execuţiei programului, pe ecran se vor tipări următoarele date:

STIVA CONTINE:

stiva
este
Aceasta

STIVA CONTINE:

este
Aceasta

Nu se poate executa "pop" dintr-o stiva goala !
STIVA ESTE VIDA !
STIVA CONTINE:

Sfirsit

La implementarea constructorului s-a putut vedea următoarea construcţie:

```
inline stack::stack(void)
{
    head=NULL;
}
```

Cuvîntul cheie "inline" este folosit pentru a indica faptul că funcţia definită astfel va fi expandată în toate locurile unde este apelată. Altfel spus, e vorba de o modalitate modernă şi elegantă de a defini macrouri-funcţie. Prefixarea cu "inline" are sens doar pentru funcţiile foarte scurte, la care este mai costisitor (în timpi de execuţie) mecanismul de apel al procedurii decît executarea corpului propriu-zis al acesteia. Un astfel de caz este şi cel de faţă: corpul funcţiei nu conţine decît o simplă iniţializare. De remarcat şi lipsa oricărei specificări de tip returnat de către constructor.

Procedura "stack::~stack" este destructorul stivei. În momentul în care domeniul de existenţă al variabilei este depăşit, se va apela această funcţie în scopul eliberării memoriei alocate dinamic.

Ar mai fi de analizat următoarea structură:


```
void stack::print_stack(void)
{
 struct _stiva *a; //var.tampon
 a= head;
 // ...
}
```

în care funcţia membru "print_stack" operează cu variabila privată "head". În caz că avem mai multe stive S1, S2 şi S3, pentru fiecare din apelurile

```
S1.print_stack();
S2.print_stack();
S3.print_stack();
```

a "head" trebuie să aibă valoarea specifică stivei în cauză. Vă veţi întreba, pe bună dreptate, cum se realizează acest lucru. Răspunsul este foarte simplu: la fiecare apel de funcţie membru se mai transmite un parametru suplimentar (invizibil utilizatorului), şi anume un pointer la obiectul în cauză. Acest pointer se numeşte "this", şi poate fi utilizat şi de către utilizator. Cel mai simplu exemplu de utilizare a lui "this" poate fi urmărit în exemplul de mai jos:

```
void stack::print_stack(void)
{
 struct _stiva *a; // var.tampon
 a= this->head; // pentru referirea variabilei membru se uti-
                // lizeaza operatorul "-> "; in acest exemplu
                // folosirea lui "this" este redundanta
 // ...
}
```

2.7. "Redefinirea" operatorilor

Termenul consacrat pentru redefinirea operatorilor este cel de "operator overloading". În cele ce urmează, vom analiza modul prin care se permite schimbarea sensului (redefinirea) operatorilor limbajului C. Utilizarea unei astfel de facilităţti duce la scrierea unor programe deosebit de inteligibile. După cum am mai arătat, faptul că se pot scrie intrucţiuni de genul

```
complex a,b,c,d,e;
...
e=(a+b)*(c-d);
```

în locul greoaielor

```
complex a,b,c,d,e;
...
```

```
e=inmulteste(aduna(a,b),scade(c,d));
```

este un avantaj care nu mai necesită alte comentarii.

Înainte de a trece la prezentarea exemplului propriu-zis, vom mai face o observaţie interesantă. Putem defini o clasă "MATRICE" care implementează operaţii cu matrici (adunări, scăderi, înmulţiri, transpuse, inverse etc.). Dacă facem acest lucru în mod "cinstit" e foarte bine. Dacă însă dorim să obţinem programe greu de descifrat, putem denumi operaţiile complet aleator. Operatorul "+" va efectua înmulţirile, "-" inversa, iar "*" adunarea. Dar atenţie: deşi acest lucru duce la scrierea unor programe nelizibile, însuşi programatorul are şanse mari să nu-şi mai poată ocoli proprile-i capcane.

În continuare vom analiza redefinirea operatorului de adunare de către o clasă "complex".

Exemplu:

```
# include <iostream.h>

class complex
{
 private:
    float real,imaginar;
 public:
    complex(int r=0,int i=1) { real=r;imaginar=i;}
    complex& operator+(complex&);
    void afiseaza_complex(void);
};

// creaza un nou numar complex care va fi initializat cu suma
// partilor reale, respectiv imaginare
complex& complex::operator+(complex& s2)
{
 complex& t= * new complex (real+s2.real,imaginar+
                            s2.imaginar);
// dupa cum am mai spus, in cadrul unei metode este permisa
// accesarea tuturor membrilor clasei; din acest motiv se pot
// face si accesari de genul "s2.real"
 return t;
}
// Tipareste partile reale si imaginare ale unui numar complex
void complex::afiseaza_complex(void)
{
 cout<<"\nComplex : "<<real<<"+i*"<<imaginar;
}
```



```
void main(void)
{
 complex a(10,11),b(1,1);
 complex& c=a+b;

 a.afiseaza_complex();
 b.afiseaza_complex();
 c.afiseaza_complex();
}
```

După execuţia programului, pe ecran se vor afişa mesajele:

```
Complex : 10+i*11
Complex : 1+i*1
Complex : 11+i*12
```

Analizînd corpul operatorului redefinit "+", remarcăm că el efectuează adunarea a două numere complexe. Sînt însumate succesiv părţile reale, respectiv imaginare. Să facem şi cîteva sublinieri:

- Pentru a putea fi returnat, rezultatului i se alocă dinamic memorie (dacă s-ar aloca static, obiectul ar fi distrus la părăsirea funcţiei).
- Apelul operatorului "+" poate avea două forme echivalente:

```
complex a,b,c;
...
a=b+c;
b.operator+(c);
```

  Se remarcă atît comoditatea, cît şi forţa sugestiei primei exprimări.

- La crearea dinamică a obiectului "complex" se va efectua şi apelul constructorului (cu parametri actuali specificaţi în listă).

2.8. FRIENDS

Exact ca în viaţă, şi în programare "prietenul la nevoie se cunoaşte". Dar despre ce este vorba? Uneori, se face simţită nevoia ca, în afara funcţiilor membru, şi alte funcţii să poată accesa varabilele private ale unei clase. Acest lucru se poate realiza prin mecanismul declarării respectivelor funcţii drept "friends" (prieteni) ai clasei respective. Astfel de declaraţii de funcţii se vor face întotdeauna în cadrul declarării clasei în cauză. În urma unei declaraţii de acest gen, "prietenii" au dreptul să acceseze datele private ale clasei respective. Facilitatea ca o funcţie să fie "prietenă" cu mai multe clase deschide un orizont foarte larg acestei idei. Iată şi un exemplu care rezolvă exact aceeaşi problemă a adunării numerelor complexe, folosind de astă dată o funcţie

"friend".

```
# include <iostream.h>

class complex
{
 private:
     float real,imaginar;
 public:
     complex(int r=0,int i=1) { real=r;imaginar=i;}
     friend complex& operator+(complex&,complex&);
     void afiseaza_complex(void);
};

complex& operator+(complex& s1,complex& s2)
{
 complex& t=*new complex(s1.real+s2.real,
                         s1.imaginar+s2.imaginar);
 return t;
}

void complex::afiseaza_complex(void)
{
 cout<<"\nComplex : "<<real<<"+i*"<<imaginar;
}

void main(void)
{
 complex a(10,11),b(1,1);
 complex& c=a+b;

 a.afiseaza_complex();
 b.afiseaza_complex();
 c.afiseaza_complex();
}
```

După execuţia programului, pe ecran se va afişa:

```
Complex : 10+i*11
Complex : 1+i*1
Complex : 11+i*12
```

În programul de mai sus, "prietenul" nu e o funcţie oarecare, ci chiar un operator redefinit. Ca şi în secţiunea "2.7", este vorba tot de un operator binar. Diferenţa faţă de secţiunea precedentă constă în faptul că, în acel

exemplu, binaritatea operatorului era oarecum mascată de către parametrul ascuns "this", pe cînd acum ea este cum nu se poate mai evidentă.

Ca ultim sfat, nu uitaţi cealaltă cunoscută zicală românească: "Fereşte-mă doamne de prieteni, că de duşmani mă feresc eu şi singur!". Este cum nu se poate mai adevărat, mai ales că funcţiile "friend" crează o breşă considerabilă în conceptul de încapsulare.

2.9. Clase derivate

Un concept important al programării OOP este cel de MOŞTENIRE. La baza lui stă un principiu foarte simplu: avînd o clasă oarecare "B" şi dorind să obţinem o altă clasă "D", asemănătoare cu prima, putem să declarăm pe "D" ca fiind DERIVATĂ din "B". În acest mod, "D" va moşteni atît variabilele cît şi funcţiile membru ale lui "B", pe lîngă care i se pot adăuga şi altele noi. Prin definiţie, "B" se va numi CLASA DE BAZĂ, iar "D" va purta numele de CLASADERIVATĂ.

Să luăm spre exemplu clasa "stack", care implementează o stivă de stringuri. Modelarea se face cu ajutorul unor liste simplu înlănţuite. Practic, exemplul din secţiunea "2.6" este copiat identic, atît că nu într-un fişier, ci în două.

Primul dintre ele se numeşte "stack.h" şi conţine declararea clasei:

```
class stack {
 protected :
     struct _stiva {
         char info[20];
         struct _stiva *next;
     } *head;     // capul stivei de stringuri

 public:
     stack(void);
     ~stack(void);

     void pop(void);
     void push(char *);
     void print_stack(void);
};
```

Unica schimbare apărută în această zonă a programului este prefixarea declarării lui "head" cu "protected" în loc de "private". Scopul acestei modificări se va vedea în curînd.

Cel de al doilea fişier va purta numele "stack.cpp" şi va arăta astfel:

```
# include <iostream.h>
# include <process.h>
# include <string.h>
# include <stdio.h>

# include "stack.h"

stack::stack(void) {
 head=NULL;
}

stack::~stack(void) {
 struct _stiva *a;     // var. tampon

 if(head!=NULL)
    do
    {
        a=head->next;
        delete head;
        head=a;
    }
    while (a!=NULL);
}

void stack::print_stack(void)
{
 struct _stiva *a;     // var. tampon

 a= head;
 if(head!=NULL)  {
    cout<<"\n STIVA CONTINE :\n";
        do {
            cout<<a->info<<"\n";
            a=a->next;
        }
        while (a!=NULL);
    cout<<"\n";
  }
  else cout<<"\n STIVA E VIDA !";
}

void stack::pop(void)
{
```



```
 struct _stiva *a;

 if (head!=NULL) {
    a=head;
    head=head->next;
    delete a;
 }
 else cout<<"Nu se poate executa \"pop\" dintr-o
          stiva goala !";
}

void stack::push(char *info)
{
 struct _stiva *a=new struct _stiva;

 if (!a) {
    cout<<"Memorie insuficienta in STACK::PUSH\n";
    exit(1);
 }
 strcpy(a->info,info);
 a->next=head;
 head=a;
}
```

Această rescriere a exemplului anterior este necesară deoarece vom specifica un nou tip de stivă, care să permită pe lîngă operaţiile prezentate anterior şi operaţia de INDEXARE. Prin indexare se va înţelege accesarea directă a elementului "i" al stivei. În cazul în care pe stivă nu există "i" elemente, se va da un mesaj de avertizare.

Dacă am lucra în TURBO C neobiectual, fie am căuta vechiul fişier şi am încerca să-i aducem modificările de rigoare, fie am rescrie pur şi simplu totul. Din cîte vom vedea, limbajele obiectuale permit abordări mult mai elegante şi mai eficiente.

Vom începe prin a crea un nou fişier, pe nume "idx_stk.h" :

```
# include "stack.h"

class i_stack:public stack {
 int max;
 public:
 i_stack(void);
 void push(char *);
 void pop(void);
```

```
};
```

În noua clasă "i_stack", care moşteneşte clasa "stack". Ce înseamnă moştenirea în cazul nostru? Pe lîngă structura lui "stack", vom mai beneficia de o variabilă privată "max" (care va păstra numărul de elemente de pe stivă) şi de un nou operator "[]" (care va realiza accesarea indexată a stivei). În plus, clasa "i_stack" va redefini procedurile "pop" şi "push", şi va avea un nou constructor, "i_stack". Prefixul "public" existent în faţa lui "stack" specifică modul în care se moştenesc datele din clasa de bază "stack": toţi membrii vor rămîne în continuare "public" sau "protected", după cum au fost declaraţi în clasa de bază. Dacă "public" ar fi lipsit, informaţiile publice şi protejate ale clasei de bază deveneau private în cea derivată.

În acest moment este timpul să facem o precizare. Toate informaţiile declarate "private" în cadrul unei clase vor exista şi în clasa derivată, dar nu vor fi accesibile. Într-un fel, acest lucru este normal. În caz contrar ar însemna că însăşi ideea de dată privată - accesibilă doar funcţiilor membru ale clasei respective - este compromisă.

Unica diferenţă între membrii "private" şi "protected" constă în aceea că ultimii vor fi accesibili şi în eventualele clase derivate.

Să analizăm în continuare fişierul "idx_stk.cpp", în care sînt implementate noile proceduri ale lui "i_stack".

```
# include <iostream.h>
# include "idx_stk.h"

void i_stack::push(char *elem)
{
 max++;          // s-a mai adaugat un element pe stiva
 stack::push(elem); // apelul vechiului "push"
}

void i_stack::pop(void)
{
 max--;          // s-a eliminat un element de pe stiva
 stack::pop();   // apelul vechiului "pop"
}

 i_stack::i_stack(void)
{
// nu mai trebuie facuta atribuirea "head=NULL;" deoarece
// inainte de a se executa constructorul clasei derivate, se va
// apela IMPLICIT constructorul clasei de baza; acesta va
// efectua atribuirea respectiva
 max=0;
}
struct _stiva* i_stack::operator[](int index)
{
 if (index<max) {
    struct _stiva *temp=head; // variabila temporara
                              // folosita la parcurgerea
                              // stivei
    for(int i=0;i<index;i++) temp=temp->next;
    return temp;
 }
 else {
    cout<<"Indexul "<<index<<" depaseste
         dimensiunea indexului maxim al stivei ( "
        <<(max-1)<<" )";
    return 0;
 }
}
```

Trebuie să subliniem că noile proceduri "push" şi "pop" le apelează pe cele vechi de îndată ce şi-au îndeplinit obligaţiile suplimentare. Modul în care se face apelul (prefixarea cu "stack::") este obligatoriu dacă se doreşte evitarea unei recursivităţi infinite. În lipsa prefixului "stack::", compilatorul nu are de unde să ştie că e vorba de metoda clasei de bază, şi va efectua un nou apel al metodei clasei derivate. Cum condiţie de oprire nu există, apelurile recursive vor continua la infinit.

În final, să studiem ultimul fişier ce face parte din "PROJECT", şi anume "main.cpp":

```
# include <iostream.h>

# include "idx_stk.h"

void main(void)
{
 i_stack my_stack;

 my_stack.push("Aceasta");
 my_stack.push("este");
 my_stack.push("stiva");

 cout<<"\n\n\n"<<my_stack[0]->info<<"\n"<<
  my_stack[1]->info<<"\n"<<my_stack[2]->info<<"\n";
```

```
 my_stack.print_stack();

 my_stack.pop();
 my_stack.print_stack();

 cout<<"\n"<<my_stack[0]->info<<"\n"<<
 my_stack[1]->info<<"\n"<<my_stack[2]->info<<"\n";

// exprimarea "my_stack[2]" va da nastere unui avertisment de
// indice prea mare

 my_stack.pop();
 my_stack.pop();
 my_stack.pop();

 my_stack.print_stack();
 my_stack.push("Sfirsit");
 my_stack.print_stack();
}
```

Procedura "main" nu are alt scop decît să utilizeze noul tip de stivă în mai multe situaţii. Modul în care are loc execuţia programului reiese din analiza mesajelor afişate pe ecran:

```
stiva
este
Aceasta

  STIVA CONTINE :

stiva
este
Aceasta

  STIVA CONTINE :

este
Aceasta
Indexul 2 depaseste dimensiunea indexului maxim al stivei ( 1 )
este
Aceasta

Nu se poate executa "pop" dintr-o stiva goala !
```



```
  STIVA E VIDA !
  STIVA CONTINE :
Sfirsit
```

Poate vi s-ar părea normal că avertizarea privitoare la depăşirea de index să fie tiparită DUPĂ afişarea celor două elemente ale stivei. Lucrurile nu stau aşa deoarece tipărirea tuturor valorilor dintr-o înlănţuire de operatori "<<" nu are loc decît în urma evaluării fiecărei expresii. Cum încercarea de evaluare a lui "my_stack[2]->info" va provoca tipărirea mesajului de avertizare, atenţionarea va fi afişată ÎNAINTEA valorilor de pe stivă.

2.10. Polimorfism

Exemplul dat în secţiunea precedentă a evidenţiat un lucru interesant, şi anume că atît clasa de bază ("stack") cît şi cea derivată ("i_stack") posedă două metode (funcţii membru) avînd acelaşi nume: "push" şi "pop". În plus, fiecare dintre ele va fi apelată DOAR de obiectele de tipul corespunzător. Marele avantaj rezidă în faptul că redeclararea unei metode nu duce la pierderea posibilităţii de a o accesa pe cea iniţială. Toate aceste proprietăţi reunite poartă numele de polimorfism.

O altă problemă direct legată de polimorfism va reieţi din exemplul de mai jos. Fişierul care îl conţine se va numi "poli.cpp" şi va înlocui în PROJECT-ul din secţiunea "2.9" fişierul "main.cpp".

```
# include <iostream.h>

# include "idx_stk.h"

void main()
{
 stack     stack1;
 i_stack   stack2;

stack *   my_stack[2];

 my_stack[0]=&stack1;
 my_stack[1]=&stack2; // un pointer la o clasa de baza B poate
                      //  pastra pointeri la clase derivate din B
 // incarcarea primei stive
 my_stack[0]->push("Aceasta");
 my_stack[0]->push("este o");
 my_stack[0]->push("stiva simpla");
```

```
// incarcarea celei de a doua stive; de notat obligativitatea
// operatorului "cast", daca se doreste apelul metodei "push"
// a clasei derivate
 ((i_stack *) my_stack[1])->push("Aceasta");
 ((i_stack *) my_stack[1])->push("este o");
 ((i_stack *) my_stack[1])->push("stiva indexata");

 cout<<"Al doilea element al stivei indexate : "
      <<(* ((i_stack *) my_stack[1]))[1]->info;

 cout<<"\n\nPrima stiva :";
 my_stack[0]->print_stack();
 cout<<"A doua stiva :";
 my_stack[1]->print_stack();
}
```

După execuţia programului pe ecran vor fi afişate următoarele date:

Al doilea element al stivei indexate : este o

Prima stiva :

STIVA CONTINE :

stiva simpla
este o
Aceasta

A doua stiva :

STIVA CONTINE :

stiva indexata
este o
Aceasta

O primă concluzie trasă în urma studierii acestui nou exemplu: întotdeauna un pointer la o clasă de bază poate păstra un pointer la o clasă derivată din ea. Atît că, dacă nu păstrăm într-o manieră oarecare informaţiile referitoare la tipul pointer-ului, nu vom putea opera corect cu metodele disponibile. Din cîte se vede, atît în cazul în care se folosesc metode redefinite, cît şi în cel al metodelor noi apărute în cadrul tipului derivat, este nevoie ca asupra pointer-ului să se aplice operatorul "cast", care să-l transforme în pointer la entitatea reală. În schimb funcţiile rămase neschimbate în cadrul ambelor clase - la noi "print_stack" - pot fi utilizate fără "casting". Mai există o

categorie specială de metode - numite VIRTUALE - care sînt capabile să deosebească pointerii la clasa derivată de cei la clasa de bază. Graţie acestei proprietăţi, ele nu vor necesita efectuarea unui "casting", nici măcar în cazul apelării unor metode redefinite în clasele derivate. Dar despre aceste metode virtuale vom vorbi în capitolul 4.

Iată-ne la sfîrşitul părţii introductive. Pînă acum, am făcut o sumară trecere în revistă a noţiunilor şi conceptelor de bază ale limbajului C++. Ea ne-a permis să avem o vedere de ansamblu a ceea ce înseamnă de fapt acest limbaj. În capitolele ce urmează, vom analiza fiecare tehnică şi concept în parte, pornind de la o abordare mai generală şi ajungînd să cunoaştem cele mai ascunse secrete.

3. CLASE

3.1. Introducere

Numele purtat de una din versiunile anterioare limbajului C++ a fost "C with Classes". Acest nume se datora noutăţii introduse: conceptul de "clasă". Pe baza claselor s-au putut dezvolta, mai apoi, toate noile tehnici de lucru. Fără clase, moştenirea şi polimorfismul nu ar fi fost decît simple speculaţii teoretice.

Conceptul de clasă este novator deoarece permite implementarea unui mecanism de definire şi utilizare a noilor tipuri de date, descrise de către utilizator. Acest lucru se face de o manieră absolut identică cu cea a tipurilor de date predefinite.

În plus, clasele permit separarea detaliilor de implementare (mecanismul intern) de cele absolut necesare pentru utilizarea noului tip de date. O excelentă analogie s-ar putea face cu modul de construcţie a unui UNIT în TURBO PASCAL: utilizatorul nu are acces decît la funcţiile şi procedurile ce apar în cadrul secţiunii INTERFACE. Cele incluse doar în IMPLEMENTATION nu pot fi utilizate de către programator. Ele pot fi apelate numai de către funcţiile din secţiunea INTERFACE, în cadrul "corpurilor" lor. O altă comparaţie sugestivă ar fi cea cu bibliotecile din TURBO C. Programatorul poate apela orice funcţie de biblioteca dacă îi cunoaşte prototipul şi efectul. El nu are însă acces la codul sursă al respectivelor funcţii.

3.2. Declararea claselor

Vom începe prin a da o serie de trei exemple, din care va reieşi un prim avantaj al utilizării claselor.

Să presupunem că scriem un program care lucrează cu corpuri în trei dimensiuni. Dacă vom memora coordonatele unui punct în trei variabile X,Y,Z şi avem o procedură ce realizează o translaţie a punctului respectiv cu TX,TY,TZ putem scrie:

Exemplul 1

```
int X,Y,Z;

void translateaza(int TX,int TY,int TZ)
{
 X+=TX;
 Y+=TY;
 Z+=TZ;
}
```


Deşi o astfel de exprimare este corectă, ea nu reflectă nicidecum existenţa unei legături între cele trei variabile X,Y,Z. Nimic nu dă de înţeles că ele au o trăsătură comună: descriu poziţia unui punct în spaţiu. Mult mai sugestivă ar fi o exprimare de genul:

Exemplul 2

```
struct coordonate_3D{
 int X,Y,Z;
};

void translateaza(struct coordonate_3D punct,
                  int TX,int TY,int TZ)
{
 punct.X+=TX;
 punct.Y+=TY;
 punct.Z+=TZ;
}
```

Cu toate că este mult mai sugestiv, şi acest mod de abordare are un inconvenient. E vorba de faptul că legătura existentă între STRUCTURA definită şi FUNCŢIA de translaţie nu este pusă în evidenţă în nici un mod. Cu alte cuvinte, se simte nevoia unui formalism care să specifice clar ideea că funcţia "translateaza" nu poate fi apelată DECÎT pentru a opera asupra unor puncte. Această problemă este uşor de remediat în C++ prin folosirea claselor.

Exemplul 3

```
class coordonate_3D{   // declararea clasei "coordonate_3D"
private :
 int X,Y,Z;    // declarare variabile membru
public :
 void translateaza(int TX,int TY,
                   int TZ);// declarare functie membru
};

// definire functie membru
void coordonate_3D::translateaza(int TX,int TY,
                                 int TZ)
{
 X+=TX;
 Y+=TY;
```

```
  Z+=TZ;
}

// declararea a trei obiecte de tipul "coordonate_3D"
class coordonate_3D obiect1,obiect2,obiect3;
```

Pentru a putea continua expunerea, e necesar să dăm cîteva definiţii.

O CLASĂ – este un tip de date definit de către utilizator, dar care se comportă întocmai ca un tip predefinit.

O INSTANŢIERE – a unei clase "CLASA" (sau un OBIECT) este o variabilă declarată ca fiind de tipul "CLASA".

O VARIABILĂ MEMBRU – este o variabilă declarată în cadrul unei clase.

O METODĂ – este o funcţie declarată (sau definită) în cadrul unei clase, şi care poate accesa toate variabilele declarate în cadrul clasei. Metodele se mai numesc şi "FUNCŢII MEMBRU".

Prin MEMBRII unei clase vom înţelege totalitatea metodelor şi variabilelor membru ale clasei.

În exemplul de mai sus avem:

a. trei obiecte de tipul "coordonate_3D" (obiect1,obiect2,obiect3);
b. trei variabile membru (X,Y,Z);
c. o singură funcţie membru ("translateaza").

Să încercăm, deci, să vedem în ce constă o clasă. O sintaxă mult simplificată arată astfel:

```
class Nume_clasa {
 [ [ private : ]  Lista_membri_1 ]
 [ [ public  : ]  Lista_membri_2 ]
};
```

Cuvîntul cheie "class" indică faptul că urmează descrierea unei clase avînd numele "Nume_clasa" (în exemplul de mai sus: "coordonate_3D"). Ca o precizare suplimentară, numele clasei poate fi orice identificator, unic în cadrul domeniului de existenţă respectiv.

Descrierea propriu-zisă a clasei constă în cele două liste de membri, prefixate eventual de cuvintele cheie "private" şi "public".

În caz că listele de membri există, ele pot conţine orice tipuri de declaraţii şi definiţii de date şi funcţii. După cum am spus deja, toate aceste date şi

funcţii vor constitui MEMBRII clasei.

Cuvintele cheie "private" şi "public" împart clasa în două secţiuni: una privată şi una publică. Să vedem care este rolul fiecăreia dintre cele două secţiuni:

public: membrii aparţinînd acestei secţiuni pot fi accesaţi de către utilizator din orice punct al domeniului de existenţă al respectivei clase.

private: membrii aparţinînd acestei secţiuni (atît date cît şi funcţii) nu pot fi accesaţi DECÎT de către metodele clasei respective. Utlizatorul clasei nu va avea acces la ei decît prin intermediul metodelor publice.

Reluînd analogia cu conceptul de "UNIT" din Pascal, se pot echivala secţiunile "PRIVATE" cu "IMPLEMENTATION" şi "PUBLIC" cu "INTERFACE".

În acest moment, dispunem de suficiente informaţii pentru a ne da seama că, în C++, încapsularea are la bază două idei:

a. cu ajutorul claselor se vor lega structurile de date şi funcţiile destinate manipulăriilor;
b. secţiunile private şi publice fac posibilă separarea mecanismului intern de interfaţă cu utilizatorul.

O sintaxă mai rafinată a claselor - şi deci mai apropiată de cea reală - ar fi următoarea:

```
Specificator_clasa Nume_clasa {
 [ [ private : ] Lista_membri_1 ]
 [ [ public  : ] Lista_membri_2 ]
};
```

Din cîte se vede, unica diferenţă faţă de definiţia precedentă constă în înlocuirea lui "class" cu "Specificator_clasa". Acesta din urmă poate fi unul dintre următoarele cuvinte cheie: "class", "struct" sau "union". Ultimele două descriu structuri de date avînd aceleaşi proprietăţi ca şi în limbajul C neobiectual, cu două modificări:

a. li se pot ataşa funcţii membru;
b. pot fi compuse din două secţiuni (una publică şi una privată).

DIFERENŢA între "class" şi "struct" constă în faptul că, dacă lipsesc specificatorii "public" şi "private", toţi membrii lui "class" vor fi IMPLICIT privaţi, iar cei ai lui "struct" publici. Şi în cazul specificatorului "union", implicit, membrii vor fi tot de tip public. Aceste observaţii lămuresc multe semne de întrebare şi dau o imagine clară a ceea ce înseamnă o clasă. Cele

spuse pînă acum vor fi subliniate de secvenţa de mai jos:

```
struct i1 {
 int a;
 double b;
 float c(void);
// a,b,c() sint IMPLICIT publici
};

union i2 {
 int a;
 double b;
 float c(void);
// a,b,c() sint IMPLICIT publici
};

class i3 {
 int a;
 double b;
 float c(void);
// a,b,c() sint IMPLICIT privati
};
```

3.3. Membrii unei clase. "Scope access operator"

Este cazul să revenim asupra celor trei exemple prezentate în secţiunea precedentă. Dacă aruncăm o privire de ansamblu, observăm că, în primul exemplu, procedura "translateaza" operează asupra variabilelor globale X, Y, Z. În al doilea exemplu, ea va opera asupra membrilor X, Y, Z ai unei structuri "coordonate_3D", transmisă ca parametru. Aceste două cazuri erau foarte simple, apelul procedurii avînd loc în manieră obişnuită a C-ului. Ultimul exemplu aduce un aspect nou: metoda "translateaza" nu poate "acţiona" independent, ci doar ca funcţie apelată de un obiect de tipul "coordonate_3D".

Din cîte se vede, nu mai este necesar ca, în cadrul funcţiei, să se precizeze "identitatea" lui X, Y şi Z. Ele vor fi considerate implicit variabile membru ale unui obiect "coordonate_3D", specificat explicit în momentul apelului. Deşi maniera exactă în care se face acest lucru o vom vedea în secţiunea următoare, dăm totuşi un exemplu care să clarifice problema. Apelul metodei "translateaza" se poate face astfel:

```
obiect1.translateaza(2,3,4);
```

În acest mod se precizează că obiectul "obiect1" apelează metoda "translateaza" cu parametrii actuali 2, 3 şi 4. Efectul acestui apel constă în adunarea

valorilor 2, 3 şi 4 la variabilele membru corespunzătoare ale obiectului "obiect1".

Să revenim la definirea lui "translateaza":

```
void coordonate_3D::translateaza(int TX,int TY,
                                 int TZ)
{
 // ...
}
```

Se remarcă imediat "prefixarea" numelui procedurii cu numele clasei urmat de "::". Acest din urmă simbol poartă numele englezesc de "scope resolution operator" sau "scope access operator", şi este utilizat în operaţii de modificare a domeniului de vizibilitate implicit. Un exemplu tipic de utilizare a sa ar fi următorul:

Exemplul 1

```
int i;

void f1(void)
{
 int i;
//...
 i++;      // se va incrementa "i"-ul local
//...
}

void f2(void)
{
 int i;
//...
 ::i++;      // se va incrementa "i"-ul global

// operatorul "::" a facilitat accesarea variabilei cu numele "i"
// aflata intr-un domeniu de vizibilitate mai inalt, si,
// in consecinta, mascata de catre variabila locala "i"
//...
}
```

Privind exemplele de mai sus, trebuie să mai facem o subliniere: în stînga lui "::" nu poate fi decît un nume de clasă sau nimic. În al doilea caz, variabila prefixată trebuie să fie globală!

Vom continua cu un sfat al lui Bjarne Stroustrup ([STRO], p.151): de cîte ori este util, se recomandă folosirea unor nume de funcţii din bibliotecile

standard ("read", "write", "open" ş.a.m.d.) drept nume de funcţii utilizator, urmînd ca primele să poată fi reutilizate în următoarea manieră:

```
//definirea functiei utilizator
int clasa::fopen(char * pNume="fisier.dat"
                 char * pAtr="rt")
{
// ...
// apelul functiei de biblioteca
 ::fopen(pNume,pAtr);
//...
}
```

În continuare, prezentăm un exemplu în care se va utiliza operatorul "scope access" ("scope resolution") pentru a opera în cadrul unei metode atît cu variabile membru, cît şi cu variabile globale.

Exemplul 2

```
int X1=10;

struct _clasa_ {
 int X1;
 void fctie(void);
} c1,c2;

void _clasa_::fctie(void)
{
 X1++;       // se va incrementa variabila membru
 ::X1++;     // se va incrementa variabila globala
}
```

Dacă avem o clasă "CLASA" şi un membru al ei numit "MEMBRU", o expresie de genul "CLASA::MEMBRU" se va numi "qualified name". Scopul lui "qualified name" este de a permite accesarea oricărui membru, chiar dacă este ascuns de o altă declaraţie de variabilă.

Exemplul 3

```
class ex {
 int contor;
public :
 void fct(void);
} ob1;

void ex::fct(void)
{
// variabila locala diferita de membrul "contor"
 int contor;
 ex::contor=100; // variabila-membru
 for(contor=0;contor<10;contor++)
//"contor"-ul lui "for" este cel local
    ex::contor+=2; // variabila-membru!!!
}
```

Încheind discuţia legată de "scope access operator", vom sublinia că prezenţa numelui clasei în faţa funcţiei membru este absolut necesară. În lipsa ei nu s-ar putea face distincţia între diversele metode purtînd nume identice şi aparţinînd unor clase diferite.

O altă motivaţie a prezenţei numelui clasei este legată de variabilele cu care se operează în cadrul funcţiei-membru. Dacă analizăm ultimul exemplu referitor la clasa "coordonate_3D", se remarcă faptul că se operează direct cu variabilele membru X, Y, Z, fără a mai da indicaţii suplimentare privind identitatea lor reală. Toate variabilele utilizate în cadrul unei funcţii-membru a clasei "CLASA" şi nedeclarate în cadrul ei, se presupun implicit a fi membri ai lui "CLASA". Dacă, în urma compilării, presupunerea se adevereşte, totul va fi în regulă. În caz contrar se încearcă identificarea lor ca variabile globale. Eşecul acestei identificări va provoca erorile de compilare corespunzătoare.

Distingerea variabilelor membru ale obiectelor din aceeaşi clasă se realizează graţie unui mecanism invizibil programatorului. El are la bază ideea de a furniza oricărei funcţii-membru un parametru ascuns, care să reprezinte un pointer la obiectul asupra căruia s-a aplicat metoda. Astfel, în Exemplul 3 al secţiunii precedente, pe lîngă cei trei parametrii declaraţi TX, TY, TZ, la fiecare apel de genul:

```
p1.translateaza(10,-2,7);
```

se mai transmite şi un pointer la obiectul "p1" (care apelează metoda). În acest fel expresiile de tipul

```
X+=TX;
```

devin

```
this->X+=TX;
```

unde, prin "this", în C++, se desemnează pointer-ul transmis ca parametru ascuns. Acest cuvînt cheie "this" este un nume de variabilă (locală funcţiilor membru), accesibilă programatorului, conţinînd un pointer la obiectul care a apelat metoda.

De fapt, rolul lui "this" nu se mărgineşte la identificarea membrilor unui obiect. După cum vom vedea în capitolele ce urmează, există situaţii de

implementare în care utilizarea lui "this" este salvatoare. Să considerăm următorul exemplu: intenţionăm să creăm o clasă "listă dublu înlănţuită". Dacă dorim să asigurăm inserarea unui element în capul listei, clasa poate arăta astfel:

Exemplul 4

```
# define NULL 0

class lista {
     int numar;
     lista * urm,* prec;
     static lista* capLista;
 public :
     void init(int i) {
         numar=i;
     }

// inserare in capul listei dublu inlantuite
 void insert(void){
     urm=capLista;
     prec=NULL;
     if(capLista) capLista->prec=this;
     capLista=this;
 }
};

// pentru a se putea face initializarea de mai jos este necesara declararea
// lui "cap_lista" ca fiind "STATIC"; pentru alte amanunte despre membrii
// statici se va studia sectiunea "3.6"
lista * lista::capLista=NULL;

void main(void)
{
 lista l1;
 lista l2;
 lista l3;

 l1.init(1);     // obiectul "l1" apeleaza functia membru "init"
 l2.init(2);
 l3.init(3);

 l1.insert();    // obiectul "l1" va fi inserat in lista
 l2.insert();    // obiectul "l2" va fi inserat in lista
```



```
 l3.insert();    // obiectul "l3" va fi inserat in lista
}
```

Alte exemple importante de utilizare a lui "this" vom regăsi în secţiunea "3.10.".

Cu obiectele aparţinînd unei clase, se poate opera întocmai ca şi cu orice tipuri de date predefinite. În secvenţa de mai jos se vor da cîteva exemple de astfel de utilizări.

Exemplul 5

```
coordonate_3D p1,p2,p3; // declararea a 3 obiecte
coordonate_3D *ptr;     // declararea unui pointer la obiect

coordonate_3D sir[20];   // declararea unui sir de 20 de obiecte
coordonate_3D * formeaza(int,int,int);
coordonate_3D * aduna(coordonate_3D,coordonate_3D);
...
p1=p2;                  // atribuire corecta
coordonate_3D=p3;       // EROARE: operatie fara sens
ptr=sir;                // corect; "ptr" va pointa la "sir[0] "
```

Din acest ultim exemplu reiese o nouă facilitate C++. În urma definirii unei clase "B", nu mai este necesară prezenţa specificatorului clasei ("class", "struct" sau "union") la fiecare declarare a unui obiect de acest tip. Va fi suficientă specificarea numelui clasei. Există însă şi o excepţie de la această regulă:

Exemplul 6

```
class litere {
 // ...
};

class numere {
 // ...
};

// ATENTIE: aceasta functie NU ESTE metoda a clasei cu acelasi nume
int numere(void)
{
 // ...
}

void main(void)
{
```

```
litere l1,l2; // CORECT: prezenta specificatorului "class"
              // este optionala
numere n1;// EROARE: clasa "numere" se suprapune ca domeniu
          // domeniu de existenta cu functia ne-membru "numere"
class numere p1,p2;    // CORECT
int numar=numere();    // CORECT: este un apel de functie!
}
```

Cu alte cuvinte, domeniul de existenţă al unui nume de clasă începe în momentul declarării ei şi se termină la sfîrşitul blocului în care a fost declarată. Unicul caz mai ciudat este cel prezentat în Exemplul 6, cînd numele clasei este identic cu cel al unui alt identificator (în cazul nostru, un nume de funcţie). În astfel de situaţii, e necesară prefixarea lui "numere" cu "class", pentru a se putea face deosebirea între clasa şi funcţia cu acelaşi nume.

Urmărind exemplul de mai jos, vom descoperi o altă situaţie specială.

Exemplul 7

```
struct nr_complex;    // DECLARATIE INCOMPLETA
nr_complex * adunare(nr_complex n1,nr_comlex n2);
//...

struct nr_complex     // DECLARATIE COMPLETA
{
 // ...
};
```

Din cîte se vede, prin DECLARAŢIE INCOMPLETĂ se înţelege declararea unei clase FĂRĂ a i se specifica lista de membri. O DECLARAŢIE COMPLETĂ a unei clase va conţine şi lista membrilor ei. Atenţie: la declararea completă a unei clase trebuie specificaţi TOŢI membrii ei. Adăugările ulterioare nu sînt posibile. Declarările incomplete sînt deosebit de utile cînd se doreşte referirea unor clase încă nedefinite.

Şi încă o observaţie, care, chiar dacă pare banală, este foarte importantă. Toate metodele unei clase vor avea exact acelaşi domeniu de existenţă ("scope") ca şi clasa căreia îi aparţin.

3.4. Accesarea membrilor unei clase.Pointeri la metode

În limbajul C, utilizarea operatorilor nu are nici un sens dacă nu li se asociază şi variabile asupra cărora să opereze.


```
int i=0,j=1,k=0;
i++;        // corect
--j;        // corect
++(--);     // EROARE: nu are nici un sens,
            // operatorii nu au operand
```

Analog, în C++, la fiecare apel de funcţie membru trebuie specificat obiectul care o apelează. De asemenea, la fiecare utilizare de variabilă membru trebuie specificat obiectul căruia îi aparţine.

Exemplul 1

```
struct cl1 {
 int a,b;
 void init(int,int);
 void inc(void);
} obiect_1,* pobiect_2;

void cl1::init(int a1,int b1)
{
 a=a1;
 b=b1;
}

void cl1::inc(void)
{
 a++;
 b++;
}

void main(void)
{
// obiectului pointat de "p_obiect_2" i se va aloca dinamic memorie
 pobiect_2=new cl1;

// membrii lui "obiect_1" vor fi initializati
 obiect_1.init(10,11);

// membrii lui "* p_obiect_2" vor fi initializati
 pobiect_2->init(10,11);

// membrii lui "obiect_1" vor fi incrementati
 obiect_1.inc();
```

```
// se va actualiza valoarea membrului "a" obiectului "obiect_1"
 obiect_1.a=pobiect_2->a+obiect_1.b;
}
```

Din cîte se vede, modul de accesare a variabilelor membru publice se face folosind operatorii "." sau "->", după cum obiectul de care aparţin este desemnat prin nume sau printr-un pointer.

Variabilele membru private nu pot fi accesate decît în cadrul metodelor clasei respective. Exemplul de mai jos va lămuri modul în care se realizează acest lucru.

Exemplul 2

```
class exemplu
{
     int numar;
 public :
     void init(int nr=0);
     void aduna(exemplu& obiect,exemplu* pobiect);
};

// Metoda "init" initializeaza membrul "numar" la valoarea
// parametrului "nr"
void exemplu::init(int nr)
{
 numar=nr;
}

// Metoda "aduna" actualizeaza valoarea membrului "numar"
void exemplu::aduna(exemplu& obiect,
                    exemplu* pobiect)
{
 numar=numar+obiect.numar+pobiect->numar;
// Membrul "numar" al obiectului ce a apelat metoda poate fi accesat di-
// rect, fara nici o prefixare. In cadrul metodei mai pot fi accesate si vari-
// abile membru private ale altor obiecte de acelasi tip, cu unica menti-
// une ca trebuie precizat si obiectul in cauza. Specificarea acestuia se va
// face printr-un nume sau printr-un pointer.
}

void main(void)
{
 exemplu obiect1,obiect2,obiect3;

 obiect1.init(2);
```



```
 obiect2.init(3);
 obiect3.init(4);

 obiect1.numar=6; // EROARE: "numar" este privat, deci
                  //  inaccesibil in functii ne-membru
 obiect1.aduna(obiect2,&obiect3); // "obiect1" apeleaza
                                  //  metoda aduna

}
```

Sintaxa apelului unei funcţii membru poate avea două forme clasice:

```
nume_obiect.nume_metoda(lista_parametri);

nume_pointer_la_obiect->nume_metoda
     (lista_parametri);
```

O a treia metodă de apelare a funcţiilor membru este cea folosind "pointerii la metode". Este vorba de un concept relativ nou (chiar şi în scurta istorie a C++), care nu apăruse în prima versiune AT&T a limbajului. Compilatorul firmei BORLAND ia însă în considerare îmbunătăţirea adusă de Stroustrup în versiunile următoare. Iată-ne, deci, faţă-n faţă cu o metodă elegantă şi eficace de manipulare a funcţiilor membru cu ajutorul pointerilor. Dacă precizăm că definirea unui "pointer la o funcţie membru a clasei B" se face prin "B::*", exemplul de mai jos nu mai are nevoie de explicaţii.

Exemplul 3

```
class clasa {
     int contor;
 public:
     void init(int nr=0) { contor=nr; }
     int increment(void) { return contor++; }
};

// tipul "pointerLaMetoda" este un pointer la o metoda a clasei
// "clasa", metoda ce nu are parametri si care returneaza "int"

typedef int (clasa::* pointerLaMetoda)(void);

void main(void)
{
 clasa c1,* pc1=&c1;
 pointerLaMetoda pM=clasa::increment;

 c1.init(1);
 pc1->init(2);
```

```
 int i=(c1.*pM)();   // operatorii ".*" si "-> *" sint utilizati la
 i=(pc1->*pM)();     // manipularea pointerilor la metode
}
```

Noul tip de dată ("pointer la metodă") desemnează adresa unei funcţii membru "M". Reamintim că metodele unei clase au IMPLICIT un parametru tip "pointer la obiect". El se va transmite "ascuns", şi va reprezenta acel "this" existent în cadrul oricărei metode. Din acest motiv, parametrul în cauză nu va mai apare în lista de parametri ai funcţiei desemnate de pointer.

Acum, avînd informaţii suficiente despre conceptul de clasă, să vedem cum ar trebui să arate o clasă ideală, în concepţia lui Bjarne Stroustrup ([STRO], p.142):

a. să aibă un set restrîns de metode publice;
b. să poată fi considerată drept o "cutie neagră", care în loc de borne de intrare şi ieşire are un set de metode publice;
c. să permită modificarea structurii datelor fără a schimba şi interfaţa (aşa-zisele borne de intrare şi ieşire din cutia neagră).

3.5. Alte amănunte privind declararea claselor. Funcţii "inline"

Să încercăm acum să sintetizăm cele spuse pînă în acest moment printr-un exemplu.

Exemplul 1

```
# include <iostream.h>

class nr_rational {
private :
 int numitor , numarator;
 void tip(void) {
    cout<<"\nNumitor="<<numitor<<",numarator="
        <<numarator;
 }
public :
 double val;
 void init(int,int);
};

void nr_rational::init(int a,int b)
{
```



```
 numarator=a;
 numitor= b!=0 ? b:1;      // se evita impartirile cu 0
 val=numarator/numitor;
 tip();
}

void main(void)
{
 nr_rational r1;
 r1.init(17,3); // corect
 r1.tip();      // EROARE: functie inaccesibila utilizatorului
 r1.val++;      // corect
 r1.numitor--;  // EROARE: variabila membru privata
}
```

Din acest exemplu reies clar proprietăţile secţiunilor private şi publice. Utilitatea lor este deosebită în cazul în care o lucrare mare este abordată modular, fiecare modul fiind tratat de către o altă persoană. Astfel, odată pusă la punct structura interfeţelor (secţiunile "public"), fiecare programator îşi poate alege sau schimba forma de reprezentare a datelor din cadrul unei clase, fără ca acest lucru să-i afecteze pe ceilalţi (deoarece clasele vor fi utilizate DOAR prin intermediul secţiunilor publice corespunzătoare).

Specificatorii "private" şi "public" pot să apară în cadrul declarării claselor de oricîte ori, şi în orice ordine. Domeniul de existenţă al acestor cuvinte cheie este din momentul întîlnirii lor, pînă în cel al apariţiei următorului specificator sau al terminării declaraţiei clasei.

Exemplu 2

```
class ex {
     int i;     // implicit "i" este privat
 public :
     int j;
     float j1; // explicit "j" si "j1" sint publice
 private :
     int k;
     float k1; // explicit "k" si "k1" sint private
 public :
     int l;     // explicit "l" este public
};
```

Există o deosebire fundamentală între "class" şi "struct" pe de o parte, şi "union" pe de altă parte. Instanţierile primelor două conţin în fiecare moment valorile TUTUROR variabilelor din lista de membri. În schimb, obiectele de

tip "union" vor păstra la un moment dat valoarea UNUI SINGUR membru din lista specificată.

Aceia dintre dumneavoastră care au folosit intensiv "union"-uri în limbajul "C" ştiu prea bine ce greoi era mecanismul de setare şi accesare a unui "union". Era nevoie de un "struct" care să conţină un "union" şi un cîmp de identificare a valorii actualmente păstrate în acesta. În C++, totul poate fi înlocuit foarte elegant de către o clasă în care:

a. mai mulţi constructori (vezi secţiunea "3.7.") se ocupă de iniţializarea diverselor cîmpuri;
b. un set de funcţii de accesare a diverselor tipuri de valori va verifica în prealabil corectitudinea operaţiei.

Ar putea să vi se pară că totul este ca şi în C, atît că, în loc să fie prezentat oarecum dezordonat, este încapsulat într-o clasă. Nu aveţi dreptate decît parţial. În C++ beneficiem în plus de o interfaţă elegantă (metodele publice), care ascunde toate detaliile şi eforturile de implementare ale mecanismului (ceea ce nu e puţin lucru).

În acest moment începem să avem o imagine destul de clară despre ceea ce înseamnă unificarea conceptelor de date şi cod. Într-un program C++ scris corect (DIN PUNCT DE VEDERE OBIECTUAL), nu au voie să existe date fără cod sau invers. Totul va fi încapsulat în cadrul claselor. Utilizarea variabilelor globale sau a funcţiilor ne-membru va însemna, în cele mai multe din cazuri, fie o concepţie greşită a programului, fie o utilizare ineficientă a programării orientate obiect. Conform manualului de utilizare al TURBO C++ ([BOR1], p.130), "datele direcţionează fluxul codului, iar acesta din urmă modelează forma şi valoarea datelor".

Şi încă o observaţie: deoarece folosirea variabilelor membru publice distruge ideea de ascundere a datelor, se spune că este bine să se evite utilizarea lor. Cei care se supun acestei restricţii implementează un set de metode care se ocupă doar cu setarea şi accesarea valorii datelor private.

Exemplul 3

```
class ex {
     int i;
 public :
     int  get_i(void) { return i; }
     void set_i(int a) { i=a;}
};
```

Putem face o scurtă paranteză, în care să arătăm că unul din dezavantajele invocate de adversarii C-ului era slaba lui tipizare. Acest lucru a dus la apariţia ANSI C, care înlătură slăbiciunea versiunii clasice. Totuşi, marea diferenţă

între ANSI C şi C++ constă în aceea că primul nu este capabil decît să semnalizeze eventualele incompatibilităţi ce apar. În C++ însă, graţie metodelor de setare şi accesare a variabilelor private, acest gen de inconveniente nu numai că este îndepărtat, dar putem spune chiar că este tratat de o manieră calitativ superioară.

O altă chestiune este legată de problema încuibăririi claselor. Dacă urmărim exemplul de mai jos, vom vedea că astfel de încuibăriri sînt permise în limbajul C++.

Exemplul 4

```
class dreptunghi {
    struct punct {
        int x,y;
        void init_punct(int x1=0,int y1=0);
    } stinga_sus,dreapta_jos;
 public :
    void init_dreptunghi(int x1,int y1,int x2,
                         int y2)
    {
        stinga_sus.init_punct(x1,y1);
        dreapta_jos.init_punct(x2,y2);
    }
};

void punct::init_punct(int x1,int y1)
{
 x=x1;
 y=y1;
}

punct p1;  // CORECT: tipul "punct" este "vizibil" in tot fisierul
dreptunghi dreptunghi_1;

void main(void)
{
 dreptunghi_1.init_dreptunghi(10,11,12,13);
}
```

Este necesară o precizare deosebit de importantă. Deşi ar părea normal ca tipul de dată "punct" să fie inaccesibil utilizatorului (fiind definit în secţiunea "private"), situaţia e de fapt cu totul alta. Regula spune astfel: definirea în cadrul unei clase B a unui nou tip de date D nu va fi influenţată de secţiunea (publică sau privată) în care are loc această operaţie.

Se mai observă că, în consecinţă, la definirea metodei de iniţializare a clasei "punct", prefixarea numelui funcţiei se face doar cu numele clasei "punct", nu şi cu cel al clasei "dreptunghi".

Există însă o excepţie ce merită amintită. În cazul declarării unui tip enumerat în cadrul unei clase, acesta va respecta regulile de accesare a secţiunii în care a fost declarat. Cu alte cuvinte, o declarare a unui tip enumerat făcută în cadrul secţiunii private nu va putea fi utilizată DECÎT de către membrii respectivei clase.

Exemplul 5

```
class ex{
    enum t1 { a , b };
 public:
    enum t2 { c , d };
};

void main(void){
 int variabila;

 variabila=ex::a; // EROARE: "ex::a" nu e accesibil
 variabila=c;     // EROARE: "c" nu poate fi identificat decit
                  // daca e prefixat cu "ex::"

 variabila=ex::c; // CORECT
}
```

Exemplul de mai sus subliniază că valorile publice de tip enumerat nu vor putea fi utilizate DECÎT prefixate cu numele clasei în care au fost declarate.

Titlul acestei secţiuni conţinea şi termenul de "funcţii inline". Să vedem despre ce este vorba:

Exemplul 6

```
# include <iostream.h>

 class coordonate_3D{
 int X,Y,Z;      // variabile membru ale clasei

// Prin DEFINIREA metodei in interiorul declararii clasei,
// functia membru devine IMPLICIT "inline" !!!

 void translateaza(int TX,int TY,int TZ)
 {
    X+=TX;
```



```
    Y+=TY;
    Z+=TZ;
 }
 void tipareste(void); // DECLARAREA unei metode
};

// Prefixarea DEFINITIEI metodei cu cuvintul cheie "inline" este
// echivalenta cu definirea functiei membru in cadrul declaratiei clasei
inline void coordonate_3D::tipareste(void)
{
 cout<<"\n\tX="<<X<<"\tY="<<Y<<"\tZ="<<Z<<"\n";
}
```

Care este noutatea introdusă de acest exemplu? În loc să ne mulţumim să declarăm procedura "translateaza" în cadrul structurii "coordonate_3D", o vom şi defini tot în cadrul clasei.

Care sînt consecinţele faptului că procedura "translateaza" va fi de tip "inline"? Acest lucru înseamnă că, practic, ea nu va fi apelată, ci expandată la locul fiecărui apel. După cum am mai spus si in capitolul 2, este vorba de un mecanism elegant de definire a unor macrouri de tipul "# define".

Definirea unei funcţii în cadrul unei clase se mai numeşte declarare "inline" implicită. Există şi o declarare explicită, care se face ca în cazul metodei "tipareste": funcţia membru e doar declarată în cadrul clasei, iar în locul în care e definită va fi prefixată de cuvîntul cheie "inline". De subliniat că orice funcţie, indiferent că este sau nu metodă, poate fi declarată inline. Atît că procedurile ce nu sînt membru a unei clase nu pot fi declarate "inline" DECÎT EXPLICIT.

Observaţii:

a. Expandarea funcţiilor "inline" va fi făcută de aşa manieră încît domeniul de existenţă al variabilelor să nu fie nicidecum afectat.

b. Faţă de cazul în care se folosesc directivele "#define", la utilizarea mecanismului "inline" se va executa şi verificarea tipului parametrilor!

c. Nu întotdeauna este posibil ca o funcţie să fie "inline". Declararea implicită sau explicită a acestei opţiuni este necesară, dar nu şi suficientă pentru ca ea să devină realitate. Este exact situaţia declarării variabilelor de tip "register": dacă nu mai sînt regiştrii disponibili, această opţiune rămîne pur teoretică.

d. ESTE INTERZISĂ FOLOSIREA în cadrul funcţiilor "inline" a STRUCTURILOR REPETITIVE DE CONTROL ("for", "while", "do while").

Vă veţi întreba, poate, la ce sînt bune funcţiile "inline"? Ele sînt deosebit de eficiente în cazurile în care transmiterea parametrilor prin stivă (operaţie

lentă) este mai costisitoare (în timp de execuţie) decît efectuarea operaţiilor din corpul funcţiei.

Facem o paranteză pentru a vă aminti că utilizatorii limbajului de asamblare care ezită între folosirea macro-urilor şi cea a procedurilor, sînt ajutaţi în luarea deciziilor de o formulă matematică destul de simplă. Astfel, dacă notăm cu "X" numărul de instrucţiuni din rutină şi cu "C" numărul de apelări a ei, putem calcula

$$F = X * C / (X + 4 * C).$$

Decizia optimă se va lua astfel: rutina se va expanda ca macrou sau nu, după cum F este mai mic sau mai mare decît 1.

Analog, firma BORLAND face cîteva recomandări, în funcţie de care să se decidă utilizarea sau nu a mecanismului "inline". Astfel, o funcţie merită să fie "inline" dacă vor fi îndeplinite simultan condiţiile:

a. corpul procedurii trebuie să conţină cel mult trei instrucţiuni;

b. numărul parametrilor şi dimensiunea lor trebuie să fie semnificativă.

Ultima recomandare a BORLAND-ului ([BOR1], p.140) sună cam aşa: utilizarea funcţiilor "inline" este eficientă doar în cazul în care codul generat de compilator pentru corpul funcţiei este mai scurt decît cel necesar apelului ei. Avînd în vedere că o astfel de comparare e imposibil de estimat printr-o simplă privire, ni se sugerează să încercăm ambele soluţii şi să o alegem pe cea optimă. Decizia optimă nu se poate lua decît în urma analizării codului obiect.

3.6. Membrii statici

Spre deosebire de "auto", "register" şi "extern", cuvîntul cheie "static" poate fi utilizat în prefixarea membrilor unei clase. Evident că, odată declarat "static", membrul în cauză va avea proprietăţi diferite de membrii obişnuiţi (ne-statici). Aceste proprietăţi apar datorită faptului că membrii statici NU APARŢIN UNUI OBIECT ANUME, ci sînt COMUNI tuturor instanţierilor unei clase.

Exemplul 1

```
class exemplu {
 int i;
 // ...
 public:

// variabila membru "contor" este declarata "public" doar in scop didactic
// (pentru a fi accesibila in functia "main")
 static int contor;      // variabila membru statica

 void inc(void) { i++; }
 void inc_contor(void) { contor++; }
 void init(void) { i=0; }
// ...

 static void functie(exemplu *); // metoda statica
} ob1,ob2,ob3;

int exemplu::contor=0; // initializarea variabilei statice

void exemplu::functie(exemplu *ptrEx)
{
 i+=76; // EROARE: nu se cunoaste obiectul ce-l "poseda" pe "i"
 ptrEx->i++; // CORECT: este specificat "proprietarul" lui "i"
 contor++;   // CORECT: "contor" este o variabila membru
             // statica, ea nu apartine unui obiect anume
}

void main(void)
{
 ob1.init();
 ob2.init();
 ob3.init();

 ob1.inc(); // ob1.i=1;
 ob2.inc(); // ob2.i=1;
 ob3.inc(); // ob3.i=1;

 ob1.functie(&ob1);       // CORECT
 exemplu::functie(&ob2); // CORECT

 functie(); // INCORECT (in afara de cazul ca exista o
            // procedura ne-membru cu acest nume)

 ob1.inc_contor();  // contor=1;
 ob2.inc_contor();  // contor=2;
 ob3.inc_contor();  // contor=3;

 exemplu::contor+=6; // contor=9;
}
```

Din exemplul de mai sus se poate vedea modul de declarare al membrilor statici, precum şi cel de iniţializare a variabilelor membru statice. Pentru fiecare instanţiere a clasei "exemplu" (ob1, ob2, ob3) va exista CÎTE O variabilă "i" DISTINCTĂ şi O SINGURĂ variabilă "contor", COMUNĂ tuturor celor trei obiecte. Comportarea variabilei "contor" poate fi cel mai bine înţeleasă analizînd funcţia "main".

Pentru o variabilă-membru statică se rezervă o singură zonă de memorie, COMUNĂ TUTUROR INSTANŢIERILOR CLASEI. Toate operaţiile ce se efectuează asupra ei au drept operand acea UNICĂ variabilă, COMUNĂtuturorinstanţierilor.

În ultima linie a lui "main" se vede un alt mod de accesare a variabilelor-membru statice. Deoarece ele nu aparţin unui obiect anume, pot fi prefixate doar de numele clasei, urmat de "scope access operator" ("::"). Deci, prefixarea variabilei membru statice cu "ob1." , "ob2." sau "ob3." nu este necesară. Ea poate fi însă folosită pentru a indica apartenenţa lui "contor" la clasa "exemplu". Întotdeauna, indiferent de prefixare, va fi incrementată aceeaşi unică va- riabilă "contor".

Din cîte se vede în exemplu, apelul metodelor statice se face exact ca şi accesarea variabilelor-membru statice. Deoarece metodele statice nu sînt apelate de un obiect anume, nu li se va transmite acel parametru ascuns "this". Deci, dacă într-o metodă statică dorim să operăm cu variabile-membru nestatice, vom fi obligaţi să furnizăm un parametru explicit de genul obiect, pointer sau referinţă la obiect.

O trăsătură specifică variabilelor-membru statice aparţinînd unor clase declarate globale este aceea că TREBUIE iniţializate în cadrul fişierului în care clasa a fost declarată.

Exemplul 2

```
class ex {
    static float a;  // DECLARARE
 public :
    static float b;  // DECLARARE
};

float ex::a=10;    // INITIALIZARE
float ex::b=12;    // INITIALIZARE
```

În cazul în care domeniul de existenţă al clasei este local, iniţializarea variabilelor-membru statice nu mai este permisă.

Toţi membrii statici sînt DOAR DECLARAŢI în cadrul clasei, ei urmînd a fi obligatoriu INIŢIALIZAŢI. În acest moment facem o observaţie

care ţine mai degrabă de istoricul produselor BORLAND decît de limbajul C++. În versiunile anterioare produsului BORLAND C++ 2.0, iniţializarea variabilelor membru statice nu era obligatorie, ca făcîndu-se IMPLICIT cu valoarea 0. La compilatoarele mai noi iniţializarea devine OBLIGATORIE, lipsa ei fiind semnalizată ca o eroare de compilare.

Principala utilizare a variabilelor-membru statice constă în eliminarea cît mai multor variabile globale ale unui program.

3.7. Constructori

3.7.1. Iniţializarea structurilor de date

Pentru a înţelege cît mai exact conceptul de "constructor" vom face o scurtă paranteză, legată de un domeniu foarte bine conturat: iniţializarea datelor. Este bine cunoscut că orice programator are grijă să nu utilizeze variabile neiniţializate. Legat de acest lucru, să vedem cum se poate face o iniţializare în C.

Exemplul 1

```
# include <string.h>

# define MAX 20

struct _adresa {
 char nume[MAX],prenume[MAX],strada[MAX];
 int numar;
} adr1={"Caragiale","Ion Luca","Pacientei",13};

void main(void)
{
 struct _adresa adr2;

 strcpy(adr2.nume,"Matei");
 strcpy(adr2.prenume,"Basarab");
 strcpy(adr2.strada,"Postei");
 adr2.numar=21;
}
```

Din acest exemplu reies foarte clar două moduri de a iniţializa structuri de date: fie în momentul declarării lor (cazul lui "adr1"), fie prin accesarea directă a diverselor cîmpuri ale variabilei ("adr2"). Cel puţin al doilea mod de lucru este deosebit de incomod, în sensul că o serie de instrucţiuni se cer repetate de un număr deranjant de mare de ori, pentru a efectua o aceeaşi succesiune

de operaţii. În plus, există riscul ca, din neatenţie, în procesul de iniţializare să fie omise unele cîmpuri.

Se poate invoca şi o altă obiecţie, mult mai serioasă: clasele care au variabile membru private nu admit iniţializări prin liste! Acest fapt este bine ilustrat de exemplul de mai jos.

Exemplul 2

```
struct s1 {
 int a;
 char c[10];
}e1={10,"abcdef"};      // CORECT

struct s2 {
 private :
    int a;
    double b;
} e2={1,2};      // EROARE

class s3 {
 int a;
 char c;
} e3={12,'c'};      // EROARE
```

De aceea, este preferabil ca iniţializările să se execute într-o procedură, aşa cum se vede în exempul următor:

Exemplul 3

```
# include <string.h>

# define MAX 20

struct _adresa {
 char nume[MAX],prenume[MAX],strada[MAX];
 int numar;
} adr1;

void init(struct _adresa* pa,char* nume,
        char* prenume,char* strada,int nr)
{
 strcpy(pa->nume,nume);
 strcpy(pa->prenume,prenume);
 strcpy(pa->strada,strada);
 pa->numar=nr;
}
```



```
void main(void)
{
 struct _adresa adr2;

 init(&adr1,"Ion","Marin","Pesterii",11);
 init(&adr2,"Bordoc","Ecaterina","Plaiului",22);
}
```

Dar şi în acest caz există o obiecţie: nu există nici o legătură implicită între procedura "init" şi structura "_adresa". Într-o abordare obiectuală, am putea elimina inconvenientul declarînd respectiva procedură ca fiind o metodă a clasei "_adresa". Cu toate acestea, s-ar face în continuare simţită lipsa unui mod elegant de iniţializare de genul:

```
int a=0,b=7;
float z=67.87;
```

Pentru a elimina aceste neajunsuri, a fost necesară implementarea unui nou mecanism, şi anume cel al constructorilor. Să începem cu un exemplu, după care vom încerca o definire riguroasă a conceptului:

Exemplul 4

```
# include <string.h>

# define MAX 20

class _adresa {
     char nume[MAX],prenume[MAX],strada[MAX];
     int numar;
 public :
     // CONSTRUCTOR!!!
     _adresa(char *,char *,char *,int);
};

_adresa::_adresa(char* nume1="",char* prenume1="",
                 char* strada1="",int nr=0)
{
 strcpy(nume,nume1);
 strcpy(prenume,prenume1);
 strcpy(strada,strada1);
 numar=nr;
}

void main(void)
{
```

```
    _adresa adr1("Bordoc","Ecaterina");
    _adresa adr2;   // daca se doreste apelul constructorului cu toti
                    // parametrii luind valori implicite, se pot omite
                    // chiar si parantezele
}
```

3.7.2. Proprietăţi specifice constructorilor

În exemplul anterior observăm că un constructor este o procedură care are acelaşi nume ca şi clasa căreia îi aparţine. Deoarece servesc la crearea şi iniţializarea obiectelor, constructorii vor fi apelaţi ori de cîte ori se crează noi instanţieri ale unei clase. Utilitatea constructorilor este evidentă cel puţin sub două aspecte:

a. constructorul asigură iniţializarea corectă a tuturor variabilelor membru ale unui obiect;
b. constructorul oferă o garanţie în plus că iniţializarea unui obiect se va efectua o singură dată.

Să notăm şi faptul că o clasă poate avea mai mulţi constructori. În principiu, aceştia se declară, se definesc şi se utilizează întocmai ca orice metodă uzuală. Totuşi, există cîteva sublinieri absolut necesare:

- Constructorii poartă numele clasei căreia îi aparţin.
- Constructorii nu pot returna valori. În plus, o convenţie face ca nici la declararea şi nici la definirea lor să nu poată fi specificat "void" ca tip returnat.
- Adresa constructorilor nu este accesibilă programatorului. Expresiile de genul "&X::X()" sînt ilegale.
- Constructorii sînt apelaţi IMPLICIT de cîte ori este nevoie.
- În caz că o clasă nu are nici un constructor declarat de către programator, compilatorul va genera implicit unul. Acesta va fi public, fără nici un parametru, şi va avea o listă vidă de instrucţiuni.
- O clasă al cărei constructor a fost generat implicit nu poate fi utilizată ca membru într-un "union".
- În cadrul constructorilor se pot utiliza operatorii "new" şi "delete".
- Constructorii pot avea parametrii luînd valori implicite.

NOTĂ: Toate aceste observaţii sintetizate mai sus sînt valabile şi pentru destructori. Aceştia din urmă vor fi analizaţi în secţiunea următoare ("3.8.").

Asemeni funcţiilor obişnuite, un constructor poate efectua orice fel de acţiuni. Tot cu titlu de observaţie amintim că modalităţile de obţinere a unui

constructor "inline" sînt identice cu cele de la funcţiile-membru obişnuite.

Trebuie să subliniem că, la orice creare dinamică a unui obiect, compilatorul va apela IMPLICIT constructorul potrivit.

Cu riscul de a ne repeta, reamintim că o clasă poate avea oricîţi constructori. Aceştia vor avea acelaşi nume, dar vor diferi prin tipul şi numărul parametrilor. Cu toate acestea, indiferent de modul în care se crează un obiect, întotdeauna compilatorul va apela constructorul corespunzător (din punct de vedere al potrivirii parametrilor).

Exemplul 1

```
# include <string.h>

# define MAX 20

struct _adresa {
 char nume[MAX],prenume[MAX],strada[MAX];
 int numar;
 _adresa(char *,char *,char *,int);
 _adresa(_adresa&);
} adr1("Ion","Marin","Pesterii",13);

_adresa::_adresa(char* nume1="",char* prenume1="",
                 char* strada1="",int nr=0)
{
 strcpy(nume,nume1);
 strcpy(prenume,prenume1);
 strcpy(strada,strada1);
 numar=nr;
}

_adresa::_adresa(_adresa& ar)
{
 strcpy(nume,ar.nume);
 strcpy(prenume,ar.prenume);
 strcpy(strada,ar.strada);
 numar=ar.numar;
}

void main(void)
{
 struct _adresa adr2("Bordoc","Ecaterina");
 struct _adresa adr3=adr1; // se apeleaza constructorul
```

```
                                    // "_adresa(adresa&)"

    adr3=adr2;      // se executa o copie bit cu bit a tuturor
                    // membrilor clasei "_adresa"
}
```

Acest exemplu îl reia pe precedentul, atît că îi mai adaugă un constructor. Noul constructor este utilizat la iniţializarea unui obiect (în cazul nostru "adr3") cu o valoare de acelaşi tip (respectiv "adr1"). Practic, instrucţiunea:

```
struct _adresa adr3=adr1;
```

va fi interpretată sub forma echivalentă

```
struct _adresa adr3(adr1);
```

La executarea instrucţiunii "adr3=adr2;" se va folosi un procedeu de copiere bit cu bit, care nu trebuie confundat cu constructorul mai sus amintit. Altfel spus, deşi cele două instrucţiuni par a se executa identic, în primul caz se va apela constructorul "adresa(adresa&)", pe cînd în al doilea nu. Acest lucru poate fi foarte bine urmărit cu ajutorul Debugger-ului încorporat, prin folosirea trasării pas cu pas. În primul caz se va intra în funcţia "_adresa::_adresa(_adresa&)", iar în al doilea nu.

Este esenţială reţinerea unei restricţii: un constructor nu poate avea ca parametru o instanţiere a clasei respective. Acesta este motivul pentru care noul constructor adăugat de noi are prototipul "_adresa(_adresa&)" şi nu "_adresa(_adresa)".

De notat că apelul constructorilor variabilelor globale se face înainte de execuţia funcţiei "main". O astfel de observaţie este deosebit de importantă la efectuarea unor operaţii care necesită anumite condiţii iniţiale. De exemplu, să presupunem ca în cadrul constructorului clasei "grafic" avem o funcţie din biblioteca grafică. Dacă modul grafic de lucru e iniţializat doar în "main", existenţa unei variabile globale de tip "grafic" va provoca o eroare de execuţie. Tot ca observaţie, amintim că apelul constructorilor variabilelor globale VA PRECEDA chiar şi pe cel al funcţiilor declarate a se executa înainte de "main" (printr-o directivă de tip "#pragma").

3.7.3. Tipuri de constructori. Ambiguităţi

În exemplul de mai jos, vom declara două tipuri de constructori:

```
class X {
// .....
 X(X&);      // COPY CONSTRUCTOR: constructor de copiere
 X(void);    // DEFAULT CONSTRUCTOR: constructor implicit
```



```
};
```

CONSTRUCTORUL IMPLICIT ("default constructor") poate fi obţinut în două moduri:

a. Prin definirea de către utilizator a unui constructor fără nici un parametru. În acest caz corpul constructorului poate conţine orice fel de instrucţiuni admise într-un context dat (în sensul că, de exemplu, dacă e vorba de un constructor "inline", el nu poate conţine instrucţiuni repetitive).
b. Prin generarea sa IMPLICITĂ de către compilator. Un astfel de constructor va fi creat ori de cîte ori programatorul declară o clasă care nu are nici un constructor. În acest caz corpul constructorului nu va conţine nici o instrucţiune.

Am crezut necesar să insistăm asupra constructorului implicit deoarece deseori au loc unele confuzii. Cîtă vreme prin acest termen va fi desemnat un constructor fără nici un parametru, indiferent că a fost creat de programator ori generat de compilator, totul va fi în regulă. În general, se greşeşte înţelegînd prin CONSTRUCTOR IMPLICIT doar una din cele două categorii amintite.

Un alt lucru deosebit de important este legat de faptul că orice clasă trebuie să posede un constructor de copiere! Pentru a creşte flexibilitatea limbajului, compilatorul va genera IMPLICIT un astfel de constructor pentru fiecare clasă la care programatorul nu a facut-o explicit. Constructorul de copiere generat implicit (a nu se confunda cu constructorul implicit) va copia membru-cu-membru toate variabilele argumentului în cele ale obiectului care apelează metoda. El va fi apelat în situaţii de genul

```
X obiect2=obiect1;
```

sub forma echivalentă

```
X obiect2(obiect1);
```

Se recomandă o mare prudenţă în îmbinarea prezenţei constructorului implicit cu cea a unor constructori cu parametrii luînd valori implicite. În caz contrar, compilatorul va semnaliza erori datorate unor ambiguităţi de genul celor din exemplul de mai jos.

Exemplul 1

```
class nr {
     int numar;
 public :
     nr(void);
     nr(int=0);
     nr(int,int=0);
```

```
};

nr::nr(void)
{
 // ...
}

nr::nr(int nn)
{
 // ...
}

nr::nr(int nn1,int nn2)
{
 // ...
}

void main()
{
 nr n0;       // AMBIGUU: constructor implicit (fara parametrii)
              // sau cu un parametru luind valoare implicita?

 nr n2(1);    // AMBIGUU: constructor cu un parametru sau cu
              // doi, din care unul luind valoare implicita?

  nr n3(1,2); // corect
 }
```

Din acest exemplu, pe lîngă problema ambiguităţii, mai trebuie reţinut un aspect. Parametrii unei funcţii-membru pot fi declaraţi ca luînd valori implicite fie la declararea metodei, fie la definirea ei. Nu se acceptă însă specificarea acestui lucru în ambele locuri!

Revenind la problema iniţializării obiectelor, vom face o nouă observaţie. Instanţierile claselor ce au constructori ne-impliciţi pot fi iniţializate în două moduri:

- Cu constructori de copiere.
- Cu constructori apelaţi cu o listă de parametrii. În acest caz există două posibilităţi echivalente de specificare a acţiunii de iniţializare:

Exemplul 2

```
class complex {
 double p_r,p_i;
 public:
    complex(int r,int i) { p_r=r; p_i=i; }
};
```


Exemplul3

```
class complex {
    double p_r,p_i;
 public:
    complex(int r,int i) : p_r(r),p_i(i) { }
};
```

3.7.4.Conversii

În continuare, va fi studiată o situaţie aparte. Este vorba de un hibrid între cazurile indicate în observaţia de mai sus. Să considerăm exemplul care urmează.

Exemplul 1

```
# include <string.h>

# define MAX 20

struct persoana {
 char numele[MAX],prenumele[MAX];
};

struct _adresa {
 char nume[MAX],prenume[MAX],strada[MAX];
 int numar;
 _adresa(persoana&);
};

_adresa::_adresa(persoana& p)
{
 strcpy(nume,p.numele);
 strcpy(prenume,p.prenumele);
 strcpy(strada,"Ploiesti");
 numar=5;
}

void main(void)
{
 persoana pers1={"Balota","Ion"};
```

```
    _adresa adr2=pers1;
}
```

Se observă că, în cadrul funcţiei "main", se încearcă iniţializarea lui "adr2" cu ajutorul unui constructor de copiere. Atît că după semnul de atribuire nu urmează obiect de tipul "_adresa", ci unul de tipul "persoana". În astfel de cazuri nu se poate recurge la constructorul de copiere (deoarece TIPURILE NU COINCID). Din acest motiv se va încerca un apel de constructor cu lista de parametri, la care primul parametru să fie de tipul "persoana&", iar restul fie că nu există, fie că iau valori implicite. Dacă există un astfel de constructor, el va fi apelat. Această operaţie poartă numele de "CONVERSIE UTILIZATOR" între tipurile "persoana" şi "_adresa".

Exemplul 2

```
class complex {
     int p_r,p_i;
 public :

// se presupune ca operatia de adunare intre 2 numere
// complexe este implementata ca metoda a clasei "complex"

     complex(int i=0 ,int j=1){
        p_r=i;
        p_i=j;
     }
}

void functie(void)
{
 complex a,b(1),c(2,3);
 a=b+1;          // se interpreteaza ca "a= b+ complex(1);"
}
```

Iată deci încă un tip de conversie definită de utilizator: din întreg (tip predefinit) în complex (tip definit de utilizator). Alte amănunte despre conversii vor fi prezentate în capitolul 5.

3.7.5. Apelul explicit al constructorilor

Înainte de a analiza secvenţa de mai jos, se impune o observaţie legată de comentariile ce o însoţesc. Fiecare comentariu conţine una sau două etichete. Aceste etichete sînt numere care indică ordinea de parcurgere a instrucţiunilor în cazul unei trasări pas cu pas cu ajutorul Debugger-ului. Altfel spus, este vorba chiar de ordinea în care calculatorul va executa instrucţiunile

programului. Am recurs la această metodă fiindcă este deosebit de expresivă din punctul de vedere al urmăririi acţiunii constructorilor. Deoarece o să mai folosim astfel de notaţii şi în alte exemple, vom face o convenţie: prin "linia 12" vom înţelege linia ce are eticheta 12, nu a 12-a linie a programului!

Exemplul 1

```
struct elem {
 int nr;
};

class clasa {
     int numar;
     struct elem * pElem;
 public:
     clasa(int);
};

clasa::clasa(int param=0)     // 3 si 8
{
 pElem=new elem;              // 4 si 9
 numar=pElem->nr=param;       // 5 si 10
}                             // 6 si 11

void main(void)               // 1
{
 clasa * pObiect;
 pObiect=new clasa(5);        // 2
 clasa Obiect=clasa(9);       // 7
 delete pObiect;              // 12
}                             // 13
```

În acest program se vor crea două instanţieri ale clasei "obiect", pentru una dintre ele alocîndu-se dinamic memorie. În ambele cazuri se va apela şi constructorul clasei respective.

În urma analizei exemplului de mai sus se impun cîteva observaţii:

a. în cadrul constructorului se poate aloca dinamic memorie (linia 4 şi 9);
b. constructorul poate fi apelat explicit în două moduri (vezi liniile 2 şi 7);
c. memoria alocată pentru obiectele create dinamic (vezi linia 2) se va elibera cu "delete" (linia 12).

3.7.6. Reducerea numărului de constructori

Există situaţii în care o clasă posedă un număr excesiv de constructori. De cele mai multe ori, aceste probleme pot fi evitate prin utilizarea judicioasă a mecanismului parametrilor luînd valori implicite.

Exemplul 1

```
// VARIANTA 1
// NUMAR EXCESIV DE CONSTRUCTORI !!!!

class ora_exacta_1 {
    int ora,minut,secunda;
 public:
    ora_exacta_1(int o,int m,
                 int s){ora=o;minut=m;secunda=s;}
    ora_exacta_1(int o,int m){ora=o;minut=m;
                              secunda=0;}
    ora_exacta_1(int o){ora=o;minut=secunda=0;}
    ora_exacta_1(void){ora=minut=secunda=0;}
};

// VARIANTA 2
class ora_exacta_2 {
    int ora,minut,secunda;
 public:
    ora_exacta_2(int o=0,int m=0,int s=0) {
       ora=o;
       minut=m;
       secunda=s;
    }
};

void main()
{
 ora_exacta_2 o1,o2(10),o3(10,10),o4(10,10,10);
}
```

3.7.7. Constructori şi obiecte statice

În cadrul acestei secţiuni trebuie făcute două observaţii:

a. Este obligatoriu ca parametrii constructorilor corespunzînd unor obiecte statice să fie EXPRESII CONSTANTE !!!

Exemplul 1

```
class ex {
 //...
 public :
    ex(int,double);
};

int f1(int a,double b,const int c)
{
 static ex ex1(2*c,2.3*c/5);// CORECT: "c" e constant
 static ex ex2(2,b);   // INCORECT: "b" e variabila
 static ex ex3(a,b);   // INCORECT: "a" si "b" sint variabile
}
```

b. Fie un program în care e definită o funcţie "F1". Să presupunem că:
   1. în funcţia "F1" este declarat un obiect static "o1";
   2. în cursul unei execuţii a programului nu se va apela niciodată funcţia "F1" (deoarece, să zicem, nu am selectat niciodată una din opţiunile meniului principal).

Se pune atunci problema: va fi sau nu va fi apelat constructorul obiectului static declarat in "F1" ? Conform specialiştilor firmei BORLAND, nu există o regulă clară care să specifice acest lucru. Ceea ce se ştie cu certitudine este faptul că, în caz de efectuare a apelului, acesta va avea loc DUPĂ cel al constructorilor corespunzînd variabilelor globale.

3.7.8. Constructorii claselor încuibărite

În exemplul de mai jos vom găsi o situaţie în care o clasă are ca membrii instanţieri ale altor clase. O astfel de situaţie de încuibărire trebuie tratată cu atenţia cuvenită, deoarece iniţializările nu sînt permise în cadrul declarării claselor. Din acest motiv există o convenţie care impune apelul constructorilor obiectelor membru.

Să considerăm o clasă "CLASA" ce conţine un obiect încuibărit de tip "OI", care, la rîndul său, are un membru "M" de tip pointer. Presupunem că în cadrul constructorului clasei "OI" se alocă dinamic o zonă de memorie la care să pointeze "M". Dacă în cadrul constructorului clasei "CLASA" se încearcă iniţializarea zonei la care pointează "M", şi constructorul lui "OI" nu a fost apelat, pointer-ul "M" va avea o valoare aleatoare, iar iniţializarea va fi ineficientă. În plus, prin această iniţializare a unei zone oarecare, s-ar putea distruge informaţii semnificative din memoria calculatorului. Pentru a evita astfel de riscuri, limbajul C++ prevede un mecanism extrem de simplu şi eficient.

Înainte de a analiza exemplul de mai jos, să vedem în cîteva cuvinte despre ce este vorba. Au fost declarate trei clase: una păstrează coordonatele unui punct şi două care definesc tipuri de vectori (definiţi prin cîte două puncte semnificative: origine şi vîrf). În continuare, vom urmări modul în care se vor apela constructorii corespunzători celor două puncte semnificative din cadrul fiecărui vector.

Exemplul 1

```
class punct {

// un punct este caracterizat de o abscisa si de o ordonata
     int x,y;

 public:
// cei doi constructori vor diferi doar prin tipul argumentelor;
// ei vor seta valorile absciseisi ordonatei
     punct(float a,float b)
     {
        x=a;
        y=b;
     }
     punct(int a=0,int b=0)
     {
        x=a;
        y=b;
     }
};

class vector1 {
  // un vector este caracterizat de doua puncte: originea si virful
     punct origine, virf;

  // variabila ce va contoriza numarul de instantieri ale clasei "vector1"
     static int   numar_vectori;
 public :
     vector1(float,float);
     vector1(int,int,int,int);
};

class vector2 {
// originea si virful vectorului vor fi specificate de cei doi pointeri
// ("pOrigine"si"pVirf")
     punct * pOrigine, * pVirf;
 public :
```



```
 vector2(int,int,int,int);
};

int vector1::numar_vectori=0;//initializarea membruluista-
                             //tic privat "numar_vectori"

vector1::vector1(float x1,float y1)
  : origine(x1,y1)
// - "origine"este initializat prin apelul explicit al constructorului avind doi
//   parametrii "float";
// - "virf" este initializat prin apelul implicit al constructorului avind doi
//   parametrii "int", ambii luind valoarea implicita 0
{
 numar_vectori++; // se incrementeaza numarul de instantieri a
                  // clasei "vector1"
}

vector1::vector1(int x1,int y1,int x2,int y2)
: origine(x1,y1) , virf(x2,y2)
// ambele puncte sint initializate prin apelul explicit al constructorului
// corespunzator
{
 numar_vectori++; // se incrementeaza numarul de instantieri a
                  // clasei "vector1"
}

vector2::vector2(int x1,int y1,int x2,int y2)
{
// se aloca memorie pentru cele doua puncte si se apeleza constructorul
// corespunzator
 pOrigine=new punct(x1,y1);
 pVirf=new punct(x2,y2);
}

void main(void)
{
 vector1 v1_1(10,10),v1_2(0,0,1,1);
 vector2 v2(16,16,17,19);
}
```

O primă observaţie ar fi legată de modul de apelare a constructorului "punct". În cadrul clasei "vector2", apelul constructorului "punct" se va face IMPLICIT în momentul în care se va ajunge la fiecare din cele două "new"-uri. Deci, în acest caz, totul decurge conform aşteptărilor.

În cazul lui "vector1" situaţia este cu totul alta. În cadrul constructorului avînd 4 parametri, se remarcă modul de apelare al constructorilor pentru cei doi membri "punct". Forma este foarte asemănătoare cu cea a iniţializării membrilor ce nu sînt obiecte. Atît că, în cazul claselor încuibărite care nu au nici constructori impliciţi şi nici constructori la care toţi parametrii pot lua valori implicite, este OBLIGATORIE această formă de iniţializare. Aşa cum se vede şi în cazul constructorului "vector1(float,float)", dacă obiectul membru are un constructor la care toţi parametrii pot lua valori implicite, această formă de "iniţializare" nu mai este obligatorie, compilatorul fiind capabil să decidă apelul constructorului la care toţi parametrii iau valori implicite. Analog s-ar petrece lucrurile şi în cazul existenţei unui constructor implicit.

După ce am înţeles ceea ce se întîmplă, să încercăm să dăm o regulă generală. În cazul în care într-un program există o clasă "CLS" care are ca membrii obiecte ale unor alte clase "CLS0", "CLS1", ..., pot apărea următoarele situaţii:

a. Dacă clasele încuibărite în cadrul lui "CLS" nu au constructori, nu va exista nici o deosebire între constructorii clasei "CLS" şi un constructor de clasă ce nu are obiecte-membru încuibărite.
b. Dacă clasele încuibărite nu au un constructor implicit şi constructor la care toţi parametrii pot lua valori implicite, este obligatorie specificarea EXPLICITĂ a numelui fiecărui obiect urmat de lista parametrilor actuali.
c. Dacă clasele încuibărite au un constructor implicit sau unul cu TOŢI parametrii avînd valori implicite, se poate proceda fie ca în cazul "a." (situaţie în care se vor apela IMPLICIT constructorii respectivi), fie că în cazul "b." (se vor specifica explicit doar parametrii doriţi). De exemplu:

```
vector1::vector1(float x1,float y1)
: origine(x1,y1),virf(10)
//...
```

unde vîrf este apelat cu un singur parametru.

Cu titlu de observaţie, subliniem că ÎNTOTDEAUNA apelul constructorilor obiectelor încuibărite se va face ÎNAINTE de executarea constructorului obiectului "cuib". Ceea ce nu se poate preciza cu exactitate este ORDINEA de apelare a constructorilor obiectelor încuibărite. Astfel, nu se recomandă scrierea unor secvenţe de genul

```
vector1::vector1(int x1,int y1,int x2,int y2)
    : origine(x1++,y1--) , virf(x1+x2,y1+y2)
{
 // ....
}
```


deoarece nu se poate stabili cu certitudine care dintre cei doi constructori "punct" va fi apelat primul.

Ca o regulă, vom preciza că specificarea constructorilor claselor încuibărite se va face OBLIGATORIU în cadrul DEFINIŢIEI constructorului "cuib", nu al declaraţiei lui.

În concluzie, putem spune că întotdeauna se vor apela constructorii claselor încuibărite, indiferent dacă modalitatea de apel va fi implicită sau explicită. În cazul în care apelul nu e explicit şi nici nu se poate face implicit (de către compilator), va fi semnalizată o eroare de compilare.

3.8. Destructori

3.8.1. Definire şi caracteristici

Destructorul poate fi privit ca şi complement al constructorului. Spuneam că, în general, constructorii sînt folosiţi pentru a aloca memorie şi pentru a efectua anumite operaţii (de exemplu, incrementarea unui contor al numărului de instanţieri ale clasei). Destructorii se utilizează îndeobşte la eliberarea memoriei alocate de primii şi la efectuarea unor operaţii inverse celor amintite mai sus (de exemplu, decrementarea aceluiaşi contor). Un alt amănunt ce indică această complementaritate ţine de modul în care se denumesc destructorii: ei poartă numele clasei precedat de un caracter "~".

```
class B {
 // ...
 public :
     B(void); // constructor
     ~B(void); // destructor
};
```

Aşa cum am afirmat şi la începutul secţiunii "3.7.2.", toate observaţiile făcute în legătură cu constructorii rămîn valabile şi în cazul destructorilor.

Exemplul 1

```
# define NULL 0

struct s{
 int nr;
 struct s * next;
};

class B {
    int i;
```

```
    struct s * ps;
 public:
    B(int);
    ~B(void);
};

B::B(int ii=0)                  // 3 si 7
{
 ps=new s; ps->next=NULL; i=ps->nr=ii; // 4 si 8
                                       // 5 si 9
}

B::~B(void)             // 11 si 14
{
 delete ps;             // 12 si 15
}                       // 13 si 16

void main(void)         // 1
{
 B * pb;
 B b=9;                 // 2
 pb=new B(3);           // 6
 delete pb;             // 10
}                       // 17
```

Să facem cîteva sublinieri pe marginea programului prezentat:

a. Destructorii se apelează implicit în două situaţii: la eliberarea cu "delete" a memoriei alocate dinamic pentru memorarea unor obiecte (linia 10) sau la părăsirea domeniului de existenţă al unei variabile (linia 17; variabila "b").

b. Acest al doilea caz merită o menţiune specială: dacă este vorba de variabile globale sau definite în "main", distrugerea lor se va face DUPĂ ultima instrucţiune din "main", dar ÎNAINTE de încheierea execuţiei programului.

c. În cazul obiectelor create dinamic, destructorul nu se va apela DECÎT dacă se încearcă eliberarea (cu "delete") zonei de memorie (linia 10).

d. Dacă avem clase încuibărite, ordinea de apelare a destructorilor este inversă celei de apelare a constructorilor respectivi.

e. Ordinea apelurilor de funcţii executate la sfîrşitul unui program TURBO C++ este următoarea:
   - funcţiile "atexit" (în ordinea inserării lor);
   - funcţiile "#pragma exit" (în ordinea codului priorităţii);
   - destructorii variabilelor globale.

   Deoarece sînt mai rar folosite, s-ar putea ca primele două categorii de

funcţii să nu vă fie familiare. În acest caz puteţi găsi informaţii amănunţite în orice manual de TURBO C.

f. Dacă în timpul execuţiei unui program se va apela funcţia "exit", nu vor fi apelaţi destructorii variabilelor locale, ci doar cei ai variabilelor globale. În caz de apel a funcţiei "abort" nu va fi apelat nici un destructor!

g. Avînd în vedere cele de mai sus, nu se recomandă utilizarea lui "exit" în cadrul destructorilor. În caz contrar se poate intra într-o recursivitate infinită.

h. Utilizatorul dispune de două moduri de a provoca apelul unui destructor:
   - prin specificarea numelui său;

```
class B {
   //...
   public: ~B();
};

void f(void) {
   B b;
   b.B::~B();// APEL DIRECT: e obligatoriu prefixul "B::"
}
```

   - prin folosirea lui "delete" (este o metodă indirectă; vezi linia 10).

3.8.2. Destructorii claselor încuibărite

În această subsecţiune se va încheia discuţia despre clasele încuibărite, începută în cadrul secţiunii "3.5" şi continuată apoi în subsecţiunea "3.7.8.". În problema apelului destructorilor claselor încuibărite se impun doar două observaţii, pe care le vom discuta pe exemplul ce urmează.

```
class baza {
    class incuib1 ob1;
    class incuib2 ob2;
    // ...
 public :
    // ...
    ~baza(void);
} ob0;
```

Care ar fi sublinierile ce se impun:

a. Apelul destructorilor obiectelor "ob1" şi "ob2" se va face doar DUPĂ execuţia destructorului "~baza".

b. În cazul în care se preferă ca în loc de obiecte încuibărite să se folosească

membrii de tipul "pointer la obiect" (să-i zicem încuibărire indirectă), va fi necesară atît alocarea dinamică, cît şi eliberarea EXPLICITĂ a memoriei necesare pentru obiectele la care vor pointa membrii în cauză. De obicei astfel de acţiuni au loc în cadrul constructorului, respectiv destructorului clasei "cuib". Dar despre acest caz particular vom discuta în secţiunea "3.10.".

3.8.3. Erori deosebite

Un prim caz care se cere studiat cu atenţie este cel din exemplul de mai jos:

Exemplul 1

```
class ex {
    double * pNumar;
 public :
    ex(double);
    ~ex();
};

ex::ex(double nr)
{
 pNumar=new double;
 *pNumar=nr;
}

ex::~ex(void)
{
 // ...
 delete pNumar;
}

void main(void)
{
 ex b1(10.66);
 ex b2=b1;              // 1
 ex b3(20.101);
 b1=b2=b3;              // 2
}
```

Liniile etichetate cu 1 şi 2 sînt generatoare de erori subtile şi greu de detectat. La baza generării lor stă mecanismul de copiere bit cu bit a obiectelor. Acesta poate fi utilizat în două situaţii: la atribuiri sau la apeluri ale constructorului implicit de copiere. Dar atenţie: eroarea nu apare la copiere, ci la apelul destructorilor, care vor fi siliţi să elibereze de mai multe ori una şi aceeaşi

zonă de memorie. Acest fapt se datorează copierii în membrii lui "b1" şi "b2" a adresei memorate în "b3". În astfel de situaţii comportarea programului poate deveni bizară, de obicei provocînd blocarea sistemului. Practic, cazul exemplului de mai sus este echivalent cu cel din secvenţa care urmează:

Exemplul 2

```
void f(void)
{
 int* pInt = new int;
 delete pInt;
 delete pInt; // GRESEALA: aceeasi zona de memorie e
              // eliberata de doua ori
}
```

3.9. Friends

În general, se consideră că există patru faze ale conceperii unui program: întocmirea caietului de sarcini, realizarea proiectului conceptual, implementarea practică şi testarea produsului final. Dacă prima fază este corect executată, va fi imposibil ca proiectarea conceptuală să nu dea naştere unor ierarhii de clase bine puse la punct. În cele mai multe cazuri, o astfel de nereuşită se datorează nepriceperii programatorului, nu inabordabilităţii obiectuale a problemei. Cu toate acestea, Bjarne Stroustrup s-a gîndit să ofere soluţii elegante pentru rezolvarea tuturor situaţiilor. În acest scop, el a creat mecanismul de "friend", care să permită abateri controlate de la ideea de protecţie a datelor prin încapsulare. Înainte de a intra în detalii, repetam un avertisment: în caz că sînteţi tentat să folosiţi acest mecanism, vă recomandăm să mai meditaţi încă o dată asupra modului în care aţi modularizat problema propusă spre rezolvare! Aveţi şanse mari fie să găsiţi un decupaj mai bun, fie să puteţi exprima "prietenia" printr-o relaţie de "moştenire" (vezi capitolul 4.). Dacă tot nu aţi izbutit, puteţi fi aproape sigur că încă nu stăpîniţi perfect conceptele OOP.

Mecanismul de "friend" porneşte de la imposibilitatea ca una şi aceeaşi metodă să fie membru a mai multor clase. Avînd în vedere acest lucru, putem examina cele două aspecte ale problemei:

a. **FUNCŢII FRIEND**

· Este vorba de funcţii care nu sînt metode ale unei clase, dar care au totuşi acces la membrii privaţi ai acesteia. Orice funcţie poate fi "friend" al unei clase, indiferent că e vorba de funcţiile ordinare, metodele altor clase sau operatori redefiniţi (vezi capitolul 5.).

Exemplul 1

```
class B {
    int a;
    int f(void);
    friend int f1(B&);
 public :
    friend int M::f2(B&,int);
}

int f1(B& rB)
{
 return rB.f();
}

int M::f2(B& rB,int j=0){
 if(rB.a>7) return j++;
      else return j--
}
```

Aşa cum se vede şi în exemplul de mai sus, nu are nici o importanţă dacă o funcţie este declarată "friend" în cadrul secţiunii "private" sau "public" a unei clase. Iată deci o excepţie: declaraţia făcută în cadrul secţiunii "private" este accesibilă "în exteriorul" clasei.

b. CLASE FRIEND

În cazul în care dorim ca toţi membrii unei clase "M" să aibă acces la partea privată a unei clase "B", în loc să enumerăm metodele lui "M" ca fiind "friends" ai lui "B", putem declara direct clasa "M" drept "friend" al lui "B". De regulă, aceste situaţii se pot evita declarîndu-l pe "M" ca fiind derivat din "B" (adică "îl moşteneşte"). Amănunte despre modul de realizare al unor astfel de derivări vom vedea în capitolul 4.

Exemplul 2

```
class M {
 // ...
};

class B {
 // ...
 friend class M;
};
```

Observaţie: relaţia de "friend" NU ESTE TRANZITIVĂ! Adică, dacă clasa B1 este "friend" al lui B2, şi B2 este "friend" al lui B3, NU înseamnă

că B1 este implicit "friend" al lui B3.

După cum reiese şi din Exemplul 1, funcţiile "friend" nu au parametrul ascuns "this" (deoarece nu sînt metode). Această carenţă se va suplini prin transmiterea unor parametrii obişnuiţi de tip pointer, obiect sau referinţă la obiect.

În încheierea secţiunii, dorim să vă prezentăm un exemplu în care folosirea funcţiilor "friend" este foarte utilă din punctul de vedere al diminuării timpului de execuţie.

Exemplul 3

```
class rational;    // declarare incompleta

class complex {
 double p_reala,p_imaginara;
 friend complex& ponderare(complex&,rational&);
public :
 complex(double r,double i) : p_reala(r),
                              p_imaginara(i) {}
 double get_real(void)     { return p_reala;}
 double get_imaginar(void) {return p_imaginara;}
};

class rational {
 int numarator,numitor;
 double val;
public :
 friend complex& ponderare(complex&,rational&);

 rational(int n1,int n2) : numarator(n1)
 {
    numitor= n2!=0 ? n2:1;
    val=((double)numarator)/numitor;
 }

 double get_valoare(void) { return val;}
};

// fiind "friend", "ponderare" are acces la membrii privati ai
// claselor "complex" si "rational"
complex& ponderare(complex& c,rational& r)
{
 complex* t=new complex(c.p_reala*r.val,
                        c.p_imaginara*r.val);
```

```
 return *t;
}

// nefiind "friend", "ponderare_ineficienta" nu are acces la
// membrii privati ai claselor "complex" si "rational"
complex& ponderare_ineficienta(complex& c,
                               rational& r)
{
 complex* t=new complex(c.get_real()
                 *r.get_valoare(),
                c.get_imaginar()*r.get_valoare());
 return *t;
}

void main(void)
{
 complex  a(2,4),b(6,9);
 rational d(1,2),e(1,3);

 a=ponderare(a,d);
 b=ponderare(b,e);

 a=ponderare_ineficienta(a,d);
 b=ponderare_ineficienta(b,e);
}
```

În exemplul ales sînt definite două funcţii de ponderare a unui număr complex prin unul raţional. Ele fac practic acelaşi lucru, însă prin metode diferite. Funcţia "ponderare" fiind "friend" a celor două clase, le poate accesa membrii privaţi, deci este foarte rapidă. Cea de a doua, "ponderare_ineficienta", nu accesează membrii privaţi ai celor două clase DECÎT prin intermediul metodelor publice. Din acest motiv timpul de acces este mai lung, iar funcţia mai lentă. Totuşi, la unul sau două apeluri de acest gen, avantajul nu e deloc evident. Dacă am dori însă ponderarea unui şir de 100000 de numere complexe, lucrurile se vor schimba net. De aceea se recomandă ca, în aplicaţiile în timp real (deci la care orice fracţiune de secundă contează: conducere de procese industriale, recunoaşteri de imagini, etc), să se folosească calea cea mai rapidă, chiar dacă aceasta nu este pur obiectuală.

3.10. Comportarea constructorilor şi destructorilor în situaţii necesitînd alocare dinamică de memorie

3.10.1. Crearea şi distrugerea şirurilor de obiecte

Începem prin a reaminti că un CONSTRUCTOR IMPLICIT este un constructor cu lista de parametrii vidă.

O restricţie oarecum supărătoare a limbajului C++ ar fi următoarea: nu putem declara un tablou de obiecte DECÎT dacă clasa respectivă are un CONSTRUCTOR IMPLICIT. Această restricţie este necesară datorită imposibilităţii de a furniza (în cadrul declaraţiei şirului) argumentele necesare constructorilor. Pentru a nu da naştere la confuzii, s-a decis ca, la iniţializarea obiectelor unui şir, să NU poată fi folosit nici măcar un constructor cu toţi parametrii luînd valori implicite.

Deoarece clasa care va da tipul tabloului poate avea ORICÎŢI constructori în afara celui implicit, ea nu-şi va pierde generalitatea. Atit că, în cazul în care respectiva clasă nu posedă un constructor implicit, trebuie făcută una din următoarele modificări:

a. eventualul constructor cu toţi parametrii luînd valori implicite va fi "rescris" (vom vedea imediat în ce mod) ca un constructor fără nici un parametru;
b. în lipsa constructorului implicit şi a celui cu toţi parametrii luînd valori implicite, se va crea un constructor implicit care să asigure minimul necesar de iniţializări pentru fiecare obiect.

Şi acum, să analizăm două exemple care vor lămuri pe deplin lucrurile.

Exemplul 1

```
// EXEMPLU ERONAT : clasa "punct" nu are constructor fara
// parametri!!!!

// In exemplul urmator vom vedea o rescriere corecta a clasei "punct".

class punct {
    int x,y;
 public :
    punct(int x1=0,int y1=0) {
        x=x1;
        y=y1;
    }
};
```

```
punct polilinie[10]; // aici apare eroarea: desi exista un
                     // constructor cu toti parametri luind
                     // valori implicite compilatorul C++
                     // refuza sa-l foloseasca
void main(void)
{
 // ...
}
```

Exemplul 2

```
// acest exemplu este rescrierea corecta a clasei "punct"
// din exemplul precedent
# include <iostream.h>

class punct {
    int x,y;
    void pct(int x1=0,int y1=0) { x=x1;y=y1; }
 public :
    punct(int x1,int y1) { pct(x1,y1); }

    punct(void) { pct(); }

// metoda de mai jos este necesara pentru initializarea
// sirurilor de obiecte de tip "punct"
    void set_xy(int x1,int y1)
    {
        pct(x1,y1);
    }
};

punct polilinie[10];

void main(void)
{
 for(int i=0;i<5;i++) {
    cout<<"\n\n TASTATI DOI INTREGI :\n ";
    int x,y;
    cin>>x>>y;
    polilinie[i].set_xy(x,y);
 }
}
```

Să analizăm, pe paşi, acţiunile efectuate în cadrul rescrierii:

a. vechiul constructor primeşte numele de "pct" şi este declarat privat;
b. se crează un CONSTRUCTOR IMPLICIT, care-l apelează pe "pct" (fără a-i furniza vreun parametru);
c. se crează un constructor avînd doi parametrii întregi, care vor fi utilizaţi ca parametrii actuali la apelul lui "pct";
d. se crează metoda "set_xy", care va servi la iniţializarea şirurilor de obiecte de tip "punct".

Din cîte se vede, funcţionalitea clasei "punct" nu este afectată, atît că îi creşte - oarecum redundant - numărul de constructori.

În încheiere, vom analiza crearea dinamică a tablourilor de obiecte. Pentru a evidenţia problemele esenţiale, am decis implementarea unei noi clase ("POLILINIE"), care să generalizeze ideea de şir de puncte.

Exemplul 3

```
# include <iostream.h>
# include <process.h>

class punct {
    int x,y;
    void pct(int x1,int y1) { x=x1;y=y1; }
 public :
    punct(int x1,int y1) { pct (x1,y1); }
    punct(void) { pct(0,0); }
};

class PoliLinie {
    int nr_pcte;
    punct * pPoliLinie;
 public:
    PoliLinie(int);
    ~PoliLinie(void);
};

PoliLinie::PoliLinie(int nr)
{
 pPoliLinie=new punct[nr]; //alocarea memoriei necesare
                           // unui sir de "nr" obiecte

// verificam daca alocarea de memorie s-a facut corect
 if(pPoliLinie==NULL)
 {
    cout<<"\n\tMemorie insuficienta !!!";
```

```
        exit(1);
    }
    nr_pcte=nr;
}

PoliLinie::~PoliLinie(void)
{
 delete[nr_pcte] pPoliLinie;// eliberarea memoriei necesa-
                            // re pastrarii unui sir de "nr_pcte"
                            // obiecte
}

void main(void)
{
 PoliLinie poli1(5);
}
```

În primul rînd trebuie remarcat modul de construire a unei polilinii: se alocă dinamic memoria necesară unui şir de "nr" puncte. Acest fapt atrage după sine o consecinţă cu totul aparte, pe care o vom discuta în cele ce urmează. Indiferent că e vorba de o instrucţiune

```
pointerPunct1=new punct;
```

sau

```
pointerPunct2=new punct[100];
```

întotdeauna "new" va returna un pointer la "punct". Din acest motiv, la întîlnirea unor instrucţiuni de genul

```
delete pointerPunct1;
delete pointerPunct2;
```

mecanismul de gestiune a memoriei alocate dinamic nu are de unde să ştie dacă "delete" se referă la eliberarea memoriei aferente unui punct sau a 100 de puncte.

Pentru a rezolva dilema, C++ oferă o soluţie elegantă (vezi destructorul "~ polilinie"). Astfel, prin construcţii de tipul:

```
delete pointerPunct1;
delete[10] pointerPunct2;
```

se va specifica EXPLICIT cantitatea de memorie de eliberat.

3.10.2. "Scurtcircuitarea" mecanismului implicit de gestionare a memoriei alocate dinamic

Spuneam că un compilator nu este capabil să deosebească un pointer la un obiect de un pointer la un şir de 1000 de obiecte. V-aţi întrebat, poate, de ce n-ar fi în stare? Răspunsul este foarte simplu: nefăcînd parte din limbajul propriu-zis, mecanismul de gestiune a memoriei alocate dinamic nu e controlat direct de compilator. Din acest motiv pot să apară erori de genul:

- iniţializarea unor zone de memorie "prin" pointeri pentru care nu s-a alocat încă memorie;
- eliberarea de mai multe ori a aceleiaşi zone de memorie.

La prima vedere, astfel de fenomene ar putea fi interpretate drept slăbiciuni ale limbajului. Dar, să nu uităm, aceste aspecte ţin de faza de execuţie a programului. În plus, nu pot fi ignorate avantajele unei libertăţi cvasi-totale de gestionare a memoriei. E adevarat că această libertate implică multă atenţie din partea programatorului, dar, în acelaşi timp, permite profesioniştilor să exploateze un calculator în maniera cea mai eficientă.

Avantajul modului deschis de a aborda problema memoriei alocate dinamic constă în faptul că permite programatorului să-şi creeze mecanisme proprii în acest domeniu. Astfel, vor putea fi eliminate slăbiciunile mecanismului standard, a cărui prea mare generalitate duce la scăderea vitezei de execuţie a programelor.

Să vedem deci, concret, despre ce este vorba. În limbajul C neobiectual, pentru gestionarea memoriei alocate dinamic, se utilizează funcţii din familia "malloc" şi "free". Modul lor de folosire vă este desigur cunoscut, precum şi faptul că, o dată apelate, programatorul pierde complet controlul a ceea ce se întîmplă în "interiorul" lor. Tot ce ştim este că, din punct de vedere al utilizatorului, comportarea acestor funcţii respectă principiile cutiei negre. Adică, deşi nu-i cunoaştem conţinutul, putem găsi o combinaţie a valorilor bornelor de intrare care să producă la bornele de ieşire valorile dorite. În cazul nostru, cutia neagră va fi funcţia "malloc", dimensiunea în octeţi a zonei de alocat va reprezenta valoarea intrare, iar pointerul la zona alocată va juca rolul de valoare regăsită la bornele de ieşire. Acest mod de abordare a problemei poate fi însă nesatisfăcător în unele cazuri.

Pentru aplicaţiile în timp real, ar fi util un mecanism care să permită un mod mai rapid de alocare a memoriei. Deoarece fiecare apel al unui "malloc" sau "free" este mare consumator de timp, în caz că se alocă frecvent memorie (să zicem de 5000 - 10000 de ori), s-ar putea cîştiga un timp preţios dacă ar exista un mecanism propriu de gestionare a memoriei. Scopul lui ar fi să evite alocarea repetată a memoriei necesare fiecărei entităţi în parte. Acest lucru

s-ar face prin alocarea simultană a memoriei necesare pentru "N" entităţi, cîştigîndu-se astfel timpul consumat la alte "N-1" apeluri. Pînă la epuizarea celor "N" entităţi alocate simultan, mecanismul nostru va acţiona astfel: de cîte ori va fi necesar, va returna cîte un pointer la una din entităţile disponibile. În plus, eliberarea unei entităţi nu va însemna eliberarea memoriei alocate pentru ea, ci adăugarea ei la setul de entităţi disponibile. Astfel, prin scurtcircuitarea mecanismului implicit de gestiune a memoriei, se înţelege înlocuirea lui cu un altul, funcţionînd pe principiile arătate mai sus.

Veţi întreba care ar fi avantajul adus de C++? Graţie constructorilor, destructorilor şi operatorilor "new" şi "delete", totul poate decurge foarte elegant şi eficace. În cadrul acestei subsecţiuni vom discuta doar cîteva principii de bază, care ne vor permite să prezentăm mai tîrziu implementarea unui adevărat mecanism de gestionare a memoriei.

Vom începe cu cîteva observaţii importante legate de variabila "this":

a. În cadrul metodelor ESTE PERMISĂ ATRIBUIREA DE VALORI POINTERULUI"this".
b. La intrarea într-un constructor, "this" va fi diferit de "NULL" DOAR DACĂ S-A ALOCAT DEJA MEMORIE PENTRU ACEL OBIECT!
c. Dacă în cadrul unui constructor SE ATRIBUIE O VALOARE VARIABILEI "this" şi, în plus, la intrarea în constructor,"this" are valoarea "NULL", înseamnă că ACEL CONSTRUCTOR A FOST APELAT DATORITĂ UNEI CREĂRI DINAMICE DE OBIECT (deci cu ajutorul operatorului "new"). Astfel, testînd la intrarea în procedură valoarea lui "this", se va şti cu exactitate tipul apelului.

```
clasa::clasa(void)
{
// pentru ca testul de mai jos sa dea rezultate corecte e OBLIGATORIU
// ca in cadrul constructorului sa existe o atribuire de tipul "this=..."
 if(this==NULL) // APEL DIN "NEW"
    // ...
 else           // OBIECTUL NU A FOST CREAT "DINAMIC"
    // ...
}
```

Înainte de a trece mai departe, să amintim un lucru pe care, pînă acum, l-am trecut cu vederea: alocarea memoriei necesare unui obiect cu ajutorul lui "malloc" NU ADUCE DUPĂ SINE APELUL ÎMPLICIT AL CONSTRUCTORULUI CLASEI RESPECTIVE !!! Acest fapt era de presupus deoarece familia de funcţii "alloc" este păstrată doar pentru compatibilite cu versiunile de C neobiectual. Deci atenţie la secvenţe ca cea de mai jos:

Exemplul 1

```
# include <alloc.h>

class ex{
    int i,j;
 public:
    ex(void){
        i=0;
        j=1;
    }
};

// ATENTIE: se lucreaza cu modelul de memorie HUGE !
void main(void)
{
 ex *p1,*p2;

 p1=(ex *) farmalloc(sizeof(ex));// nu se va apela
                                 // constructorul, ci se
                                 // va aloca doar memorie

 p2=new ex; // se aloca memorie si se apeleaza constructorul
}
```

Revenind la ideea precedentă, să mai facem o observaţie: dacă în cadrul unui constructor se face o atribuire către "this", ORICE MANIPULARE A UNOR MEMBRI AI CLASEI ÎNAINTEA ATRIBUIRII POATE AVEA EFECTE DEZASTRUOASE!

Exemplul 2

```
class clasa {
    int i,j;
 public :
    clasa(void);
};

clasa::clasa(void)
{
 i=16; // POSIBIL DEZASTRU: in caz ca "this==NULL" nu se
       // stie ce zona de memorie va fi initializata!

 if(this==NULL) this=aloca_memorie();
```

```
 i=20;  //  In acest loc "this" difera sigur de NULL. Zonele de memo-
 j=23;  //  rie initializate vor fi cele rezervate pentru membrii clasei.
}
```

Din păcate, limbajul C++ nu oferă implicit vreo metodă prin care destructorul unei clase să poată deduce dacă obiectul a fost sau nu creat dinamic. În caz că această informaţie este necesară, clasei i se poate adăuga un membru care să indice acest lucru (vezi exemplul 5). De asemenea, nu există nici o modalitate implicită de a afla dacă un destructor este apelat datorită unui "delete" sau datorită părăsirii domeniului de existenţă a unei variabile.

În schimb, există o metodă graţie căreia poate fi evitat mecanismul standard de eliberare a memoriei: ASIGNAREA VALORII "NULL" POINTERULUI "THIS" !!!! Astfel, dacă într-un destructor apelat de "delete" vom face asignarea "this=NULL", zona de memorie la care pointează "this" nu va mai fi eliberată de către gestionarul standard al memoriei.

Un alt aspect demn de reţinut este legat de modul de alocare şi eliberare a memoriei în cazul unor obiecte create dinamic şi care au ca membrii pointeri la diverse entităţi. Trebuie precizat că, în momentul creării obiectului în cauză, se va aloca memorie DOAR pentru membrii săi, NU şi pentru eventualele entităţi la care ar putea pointa aceştia. Pentru aceste entităţi este necesară o alocare EXPLICITĂ de memorie, eventual chiar în cadrul constructorului. Din acest motiv, eliberarea necesarului de memorie corespunzător unor astfel de entităţi se va face tot EXPLICIT, de obicei în cadrul destructorului clasei. Să analizăm un exemplu practic:

Exemplul 3

```
class CLASA {
     int i;
     CLASA * pCls;
 public :
     CLASA();
     ~CLASA();
};

CLASA::CLASA()
{
// pCls=new CLASA;
//  EROARE: s-ar provoca un apel recursiv infinit al constructorului

// urmeaza varianta corecta de alocare a memoriei pentru
// entitati "pointate" de membri ai unui obiect

 pCls=(CLASA *) new char[sizeof(CLASA)];
```



```
// s-au mai alocat 6 octeti pentru obiectul pointat de "pCls"

 i=0;
 pCls->i=6;
}

CLASA::~CLASA()
{
// delete pCls;
//  EROARE: s-ar provoca apelul recursiv infinit al destructorului

// urmeaza varianta corecta de eliberare a memoriei necesare
// unor entitati "pointate" de membri unui obiect

 delete[sizeof(CLASA)] (char *) pCls;

// s-au eliberat cei 6 octeti la care pointa "pCls"
}

void main()
{
 CLASA * pC;

 pC=new CLASA;  //  se aloca 6 octeti pentru obiectul nou creat
 delete pC;  //  se elibereaza cei 6 octeti ocupati de obiect
}
```

Există totuşi o posibilitate de a aloca simultan memorie atît pentru un obiect, cît şi pentru eventualele entităţi la care pointează unii membrii ai săi. Iată cum se poate realiza acest lucru:

Exemplul 4

```
# include <iostream.h>
# include <process.h>

class CLASA {
     int i;
     CLASA * pCls;
 public :
     CLASA();
};

CLASA::CLASA()
{
```

```
if(this!=NULL) {
   cout<<"EROARE : constructorul nu a fost apelat
                   cu NEW";
   exit(1);
}

// se va aloca memoria necesara pastrarii a doua obiecte de tip "CLASA".
// "This" va pointa la primul dintre ele.
 this=(CLASA *) new char[2*sizeof(CLASA)];

 pcls=this+1; // "pCls" va pointa la cel de al doilea obiect de
              // tip "CLASA"

 i=0;                  // se seteaza "this -> i"
 pCls->i=5;            // se seteaza "pCls -> i"
}

void main()
{
 CLASA * pC;

 pC=new CLASA; // la apelul constructorului se vor aloca 12 octeti

 delete pC;// se vor eliberat TOTI cei 12 octeti alocati
}
```

Deosebirea este uşor sesizabilă. În Exemplul 3 se utiliza alocatorul standard de memorie de două ori: mai întîi pentru obiectul propriu-zis, şi apoi pentru cel la care va pointa membrul său "pCls". În Exemplul 4 se va utiliza o singură dată gestionarul memoriei. Dar, de data aceasta, se alocă memorie ÎN ACELAŞI TIMP atît pentru obiect, cît şi pentru entitatea la care pointează "pCls". Datorită acestei stratageme, nu va mai fi necesară prezenţa unui destructor, deoarece se va şti exact cîtă memorie a fost alocată în cadrul constructorului. În momentul întîlnirii lui

```
delete pC;
```

se va elibera întreaga cantitate de memorie.

În încheierea acestei subsecţiuni, vom prezenta un exemplu de utilizare a majorităţii tehnicilor despre care s-a vorbit pînă acum. Vom începe prin a explica pe scurt ce se întîmplă în cadrul programului. S-a implementat o clasă "persoana", ale cărei instanţieri pot fi create în trei moduri diferite. Pentru fiecare mod în parte, constructorul va executa acţiuni specifice, după cum urmează:

a. OBIECT_CREAT_STATIC:

Este vorba de crearea unor obiecte pentru care nu se va aloca dinamic memorie.

b. OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE:

Este vorba de o simulare a tipului de dată "REFERINŢĂ". Pointer-ul "this" al unui obiect "OB" va fi setat la o valoare reprezentînd un pointer la un obiect "OB1" de acelaşi tip, creat deja. În acest mod, orice operaţie efectuată asupra unui membru al lui "OB" va modifica membrul corespunzător al obiectului "OB1", referit de către obiectul "OB". Cu alte cuvinte, "OB" va fi un nou nume al obiectului "OB1".

c. OBIECT_CREAT_DINAMIC_CU_ALOCARE_DE_MEMORIE:

Acest mod de creare corespunde situaţiei în care memoria necesară obiectului în cauză nu va fi alocată decît în cadrul constructorului în cauză.

Exemplul 5

```
// Programul de mai jos este compilat cu modelul de memorie "HUGE".
// Pe parcursul lui se va utiliza functia de biblioteca "farcoreleft" pentru
// a se cunoaste cantitatea de memorie disponibila in "heap".

# include <iostream.h>
# include <process.h>
# include <string.h>
# include <alloc.h>

enum mod_de_creare {
 OBIECT_CREAT_STATIC ,
 OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE ,
 OBIECT_CREAT_DINAMIC_CU_ALOCARE_DE_MEMORIE
};

class persoana {
     char nume[20];
     char a[1000];
     int  virsta;

     mod_de_creare indicator;
 public:
     persoana(mod_de_creare,char *,int,persoana *);
     ~persoana(void);
};
```

```
// Creaza in unul din cele 3 moduri posibile un obiect de tip persoana
persoana::persoana(mod_de_creare modCreare,
 char *pNume,int _virsta, persoana *pPersoana=NULL)
{
 switch(modCreare) {
 case OBIECT_CREAT_STATIC :
//initializari
    if(this!=NULL) {
       strcpy(nume,pNume);
       virsta=_virsta;
    }
    else {
       cout<<"EROARE: obiect static cu this==
             NULL";
       exit(1);
    }
    break;

 case OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE :
    if(this==NULL)
       if(pPersoana!=NULL)
       {
          this=pPersoana;

          strcpy(nume,pNume); // initializari
          virsta=_virsta;
       }
       else {
// NU EXISTA "PERSOANA" LA CARE SA SE FACA
// "LEGATURA"
          cout<<"EROARE : this ramine NULL\n";
          exit(1);
       }
     else {
// CONSTRUCTORUL NU A FOST APELAT DE UN "NEW"
       cout<<"EROARE : this nu este NULL\n";
       exit(1);
    }
break;

 case OBIECT_CREAT_DINAMIC_CU_ALOCARE_DE_MEMORIE :
    if(this==0){
// Urmeaza alocarea memoriei necesare si testarea corectitudinii
```



```
// acestei operatii.
       this=(persoana *)
              new char[sizeof(persoana)];
       if(this==NULL) {
          cout<<"MEMORIE INSUFICIENTA\n";
          exit(1);
       }
//initializari
       strcpy(nume,pNume);
       virsta=_virsta;
       }
       else {
// CONSTRUCTORUL NU A FOST APELAT DIN "NEW"
          cout<<"EROARE : this nu este NULL\n";
          exit(1);
        }
       break;
}

// NOTA: - initializarea ce urmeaza nu se putea face la inceputul metodei
//          deoarece in doua din cazuri "this" ar urma sa-si modifice va-
//          loarea
//        - de remarcat ca in cazul in care obiectul este creat ca
//   OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE,
//          optiunea respectiva va fi inscrisa in cimpul "indicator" al
//          obiectului indicat de "pPersoana" !
 indicator=modCreare;
}

persoana::~persoana(void)
{
// Nu se permite eliberarea memoriei decit pentru obiectele la care nu
// exista "referinte". In cazul obiectelor "referite" de alti pointeri, s e
// scurtcircuiteaza mecanismul standard de eliberare a memoriei (prin
//  "this=NULL").

 if(indicator==
      OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE)
    this=NULL;
}

unsigned long m;
```

```
void mn(void)
{
 persoana p1(OBIECT_CREAT_STATIC,"ion",24);

 persoana *p2 = new persoana(
        OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE,
           "p",34,&p1);

 persoana *p3 = new persoana(
       OBIECT_CREAT_DINAMIC_CU_ALOCARE_DE_MEMORIE,
         "v",55);

 persoana *p4 = new persoana(
        OBIECT_CREAT_DINAMIC_FARA_A_ALOCA_MEMORIE,
          "a",43,p3);

 persoana *p5 = new persoana (
        OBIECT_CREAT_DINAMIC_CU_ALOCARE_DE_MEMORIE,
         "o",79);

 m=farcoreleft();      // m=60288;

 delete p2;
 m=farcoreleft();      // m= 60288; nu s-a eliberat memoria
                       // alocata deoarece "p2" refera pe "p1"

 delete p3;
 m=farcoreleft();      // m= 60288; nu s-a eliberat memoria
                       // alocata deoarece "p3" e referit de "p4"

 delete p4;
 m=farcoreleft();      // m= 60288; nu s-a eliberat memoria
                       // alocata deoarece "p4" refera pe "p3"

 delete p5;
 m=farcoreleft();      // m= 61328; s-a eliberat memoria
                       // alocata deoarece "p5" nu e referit si nu
                       // refera pe nimeni
}

void main(void)
{
```



```
 m=farcoreleft(); // m=62368;
 mn();
 m=farcoreleft();          // m= 61328;
}
```

Utilizarea unor artificii de genul "OBIECT_CREAT_FARA_ALOCARE_DINAMICA_DE_MEMORIE" trebuie făcută cu foarte mare atenţie, pentru simplul motiv că ele pot scăpa uşor de sub control, avînd spectaculoase efecte distructive. De exemplu, un obiect "O1" referit de "O2" şi "O3" poate fi şters din greşeală, fără a efectua o necesară operaţie de "resetare" a referinţelor la el ("O2" şi "O3"). Dacă, mai tîrziu, vom folosi obiectul "O2", fie se vor citi date eronate, fie se vor modifica locaţii de memorie de la adrese aleatoare.

În cadrul unor programe scurte, scrise de un singur programator, riscurile sînt destul de reduse. La proiecte de dimensiuni mai mari şi la care colaborează mai mulţi programatori, lipsa de coordonare şi consecvenţă poate duce la erori greu de localizat.

Avînd în vedere cele enumerate, vă sugerăm să luaţi exemplul de mai sus drept o simplă gimnastică a minţii. În caz de nevoie, recomandăm utilizarea tipului "referinţă" existent în limbajul C++. Dar aceasta nu înseamnă să nu utilizaţi tehnicile prezentate în cadrul subsecţiunii ori de cîte ori vi se pare necesar.

3.11. Probleme rezolvate.Probleme propuse spre rezolvare

3.11.1. Array-uri de dimensiuni variabile

Mulţi dintre cei care au făcut trecerea de la BASIC la C au avut o vreme nostalgia declarării unor şiruri de dimensiuni variabile, în funcţie de necesităţi. Înainte de a studia modul de abordare a unei astfel de probleme în C++, trebuie să vă facem o recomandare: ar fi bine ca, în urma parcurgerii întregii cărţi, să reveniţi asupra acestui exemplu. Am ţinut să vă atragem atenţia asupra acestui fapt deoarece doar în momentul în care veţi stăpîni şi concepetele de polimorfism şi redefinire a operatorilor vă veţi putea da seama de diferenţa reală între programul de mai jos şi echivalentul său neobiectual. Noul tip de dată se va numi BASIC_ARRAY (în amintirea limbajului după care ne-am inspirat) şi va avea trei metode:

- un constructor (crează şi intializează array-urile);
- un destructor (eliberează memoria alocată pentru un array);

– o funcţie de concatenare a două array-uri (exemplu: dacă avem a={1,2,3} si b={4,5,6}, în urma concatenării lui "a" cu "b" vom obţine a={1,2,3,4,5,6} şi. b={4,5,6}).

Să analizăm acum programul:

```
# include <iostream.h>
# include <process.h>
# include <conio.h>

# define nl "\n"

class BASIC_ARRAY {
 int * pointer;      // pointer la array
 int dimension;      // dimensiune array
 void print(void);   // tiparire array
 void test(void *); // testeaza daca s-a alocat corect memorie
public:
 BASIC_ARRAY(int); // constructor

 ~BASIC_ARRAY(void)    // destructor
  {
    delete[dimension] pointer;
  }

// procedura de concatenare
     void concatenate(BASIC_ARRAY&);
};

// Testeaza valoarea unui pointer pentru care s-a alocat dinamic memorie
// Daca pointer-ul este setat la NULL inseamna ca operatia de alocare
// dinamica a memoriei nu s-a efectuat corect.
void BASIC_ARRAY::test(void * ptr)
{
 if(ptr==NULL)
 {
     cout<<"\n\n\t MEMORIE INSUFICIENTA";
     exit(1);
 }
}

// Creaza un array de intregi avind "nrElem" elemente.
// Initializeaza array-ul cu valori citite de la tastatura.
BASIC_ARRAY::BASIC_ARRAY(int nrElem)
```



```
{
// mesaj explicativ
 cout<<nl<<nl;
 cout<<"Tastati "<<nrElem<<" intregi !"<<nl;

// alocarea memoriei necesare array-ului
 dimension=nrElem;
 pointer=new int[nrElem];

 test((void *) pointer);

// initializare array
 for(int i=0;i<nrElem;i++) cin>>pointer[i];

// tiparire array
  print();
}

// Tipareste un array
void BASIC_ARRAY::print(void)
{
 cout<nl<<nl<<"Sirul este : ";

 for(int i=0;i<dimension;i++)
    cout<<pointer[i]<<" ";
 cout<<nl;

 cout<<"Apasati o tasta !"<<nl;
 getch();
}

// Concateneaza array-ul care a apelat metoda cu cel primit ca
// parametru ( PRIN REFERINTA ! ); rezultatul se va pastra "in"
// obiectul care a apelat metoda

void BASIC_ARRAY::concatenate
                    (BASIC_ARRAY& refARRAY)
{
 int *tmp;
 int final_dimension;

// calcularea noii dimensiuni a rezultatului
 final_dimension=dimension+refARRAY.dimension;
```

```
 // alocare memorie necesara
  tmp=new int[final_dimension];

  test((void *) tmp);

 // copierea celor doi "operanzi" in cadrul "rezultatului"
 for(int i=0;i<final_dimension;i++)
  if(i<dimension) tmp[i]=pointer[i];
  else tmp[i]=refARRAY.pointer[i-dimension];

  delete[dimension] pointer; // eliberarea memoriei folosite
                             // de primul operand

  dimension=final_dimension; // setare dimensiune corecta

  pointer=tmp; // setare "pointer la array"

 // tiparire rezultat
  print();
}

void main(void)
{
 BASIC_ARRAY a(3);
 BASIC_ARRAY b(4);

 int i;
 cout<<"Tastati dimensiunea sirului : ";
 cin>>i;
 cout<<nl;

 BASIC_ARRAY c(i);

 a.concatenate(b);
 a.concatenate(c);
}
```

Ca exerciţii propuse spre rezolvare, vă sugerăm să completaţi lista metodelor clasei cu următoarele funcţii:

- "**int find**(BASIC_ARRAY& model)":
  Caută în array-ul iniţial prima apariţie a array-ului "model". Returnează 1 sau 0 după cum modelul există sau nu în cadrul array-ului iniţial.

- "**int replace**(BASIC_ARRAY& model_vechi,
  BASIC_ARRAY& model_nou)":
  Înlocuieşte în cadrul array-ului iniţial toate apariţiile lui "model_vechi" cu "model_nou". Returnează numărul de substituţii efectuate.
- "**int eliminate**(BASIC_ARRAY& model)":
  Elimină din array-ul iniţial toate apariţiile array-ului "model". Returnează 1 sau 0 după cum eliminarea a fost sau nu posibilă.

3.11.2. HASH-TABLE pentru numere de telefon

În această secţiune vom prezenta un concept utilizat deseori de către informaticieni: "hash tables" ("tabele de hashing"). În câteva cuvinte, despre ce este vorba: fie o mulţime de elemente păstrată în memoria calculatorului ca o listă neordonată. A afla dacă o valoare anume este inclusă în această mulţime înseamnă a parcurge SECVENŢIAL lista şi a testa element cu element. Dacă se folosesc diverşi algoritmi de sortare a mulţimii, căutarea se poate face mult mai rapid. Dacă nici astfel viteza nu este suficient de mare, se va îmbina sortarea cu "hashing"-ul.

Ideea de bază în "hashing" este de a încerca să împărţim elementele mulţimii într-un număr oarecare de clase de echivalenţă (având fiecare cam acelaşi număr de elemente - statistic vorbind). Dacă reuşim acest lucru, înseamnă că la căutarea unui element în cadrul mulţimii vom stabili în prima fază cărei clase de echivalenţă îi aparţine, iar apoi îl vom căuta doar în cadrul acelei submulţimi. În cazul în care clasele de echivalenţă vor fi sortate, viteza creşte şi mai mult.

Acestea fiind spuse, să vedem ce-şi propune programul de mai jos. Se doreşte implementarea unei agende telefonice în care căutarea numerelor de telefon să se facă printr-o tabelă de hashing. Criteriul împărţirii în clase de echivalenţă va fi cifra finală a numărului de telefon. În concluzie, vom avea 10 clase de echivalenţă, având aproximativ acelaşi număr de elemente. Tabela de hashing este constituită dintr-un şir de 10 elemente, fiecare conţinând un pointer la o lista simplu înlănţuită. Toate elementele ce aparţin aceleiaşi liste au în comun faptul că se sfîrşesc cu aceeaşi cifră. Deoarece acest exemplu are doar un scop didactic, clasele de echivalenţă nu vor fi sortate. Deci, a căuta în "agendă" un număr de telefon ce se termină cu cifra "i" înseamnă a-l căuta secvenţial în cadrul listei la care pointează al "i"-lea element din tabela de hashing.

Ca exerciţiu, vă propunem spre rezolvare ordonarea claselor de echivalenţă şi căutarea (inserarea) în cadrul acestora prin metoda "divide and conquer" (metoda "înjumătăţirii intervalului"). De asemenea, vă propunem scrierea

unui destructor al clasei "hash_array".

```
# include <stream.h>
# include <process.h>
# include <string.h>
# include <conio.h>

# define ON  0
# define OFF 1

# define nl "\n"

class hash_array {

 struct elem {
     char nume[20];      // nume persoana
     long nr_telefon;    // numar de telefon atasat
     elem *urm;          // pointer la urmatorul element

     elem(long,char*); // constructor element

 }  * sir[10];          // tabela de hashing

 void tipareste(void);      // tiparire agenda telefonica

 void test(void*); // testeaza corectitudinea alocarii
                   // dinamice de memorie
public:
 hash_array(void);  // constructor

// cautare element in agenda
 elem* cauta_numar(long numar_telefon,
                   char *nume="",int insereaza=OFF);

// inserare element in agenda
 elem* insereaza(long numar_telefon,char * nume) {

     // in urma cautarii, elementul se va insera (optiunea ON)
     elem* pElem=cauta_numar(numar_telefon,
                             nume,ON);

     tipareste();
     return pElem;
 }
};

// Initializare element multime
elem::elem(long n,char * p)
{
 nr_telefon=n;
 strcpy(nume,p);
 urm=NULL;
}

// Initializare tabela de hashing (nu exista inca elemente)
hash_array::hash_array(void)
{
 for(int i=0;i<10;i++) sir[i]=NULL;
}

// Testeaza valoarea unui pointer; daca acesta este NULL inseamna ca
// ultima operatie de alocare dinamica de memorie nu s-a efectuat corect
void hash_array::test(void * pointer)
{
 if(pointer==NULL)
 {
     cout<<"\n\n\tMEMORIE INSUFICIENTA !!!";
     exit(1);
 }
}

// Cauta in agenda un numar de telefon . Daca nu-l gaseste si
// parametrul "insereaza" are valoarea "ON", il va insera
elem* _hash_array::cauta_numar(long numar_telefon,
                               char * nume, int insereaza)
{
 elem* tmp,*cap_lista;
 int flag=OFF;          // numarul nu a fost inca gasit

 // determinare "intrare" in tabela de hashing
 int indice=numar_telefon-(numar_telefon/10)*10;
 cap_lista=tmp=sir[indice];

// cita vreme numarul nu a fost gasit si mai exista elemente
// netestate se continua cautarea
 while (tmp!=NULL && flag!=ON)
```

```
    if(tmp->nr_telefon!=numar_telefon)
      tmp=tmp->urm;
    else {
        flag=ON;      // numar gasit
        if(insereaza==ON) {
        cout<<nl<<nl<<nl
         <<"Numarul "<<numar_telefon<<"nu poate fi"
         <<" inserat de doua ori !"<<nl<<nl<<nl;
        return NULL;
    }
}

// daca inscrarea este ceruta si posibila, ca se va face in
// capul listei (ultimul adaugat va fi chiar "intrarea" in tabela)
 if( insereaza==ON ) {
    sir[indice]=new elem(numar_telefon,nume);

    test((void *) sir[indice]);

    sir[indice]->urm=cap_lista;
    return sir[indice];
  }

 return tmp;
}

// Tipareste intreaga agenda telefonica
void hash_array::tipareste(void)
{
 cout<<nl<<nl<<nl<<"AGENDA DE TELEFON : "<<nl<<nl;

 for(int i=0;i<10;i++) {
//  pentru fiecare "intrare" in tabela de hashing ...

    cout<<"Numere ce se sfirsesc cu "<<i
       <<" :"<<nl;
    elem* tmp=sir[i];

    while(tmp!=NULL) { // ... se tipareste clasa de echivalenta
        cout<<tmp->nr_telefon<<" "<<tmp->nume<<nl;
        tmp=tmp->urm;
    }
 }
```



```
 getch();
}

void main(void)
{
 hash_array agenda;

  agenda.insereaza(1110,"Manu");
  agenda.insereaza(1111,"Condas");
  agenda.insereaza(1112,"Gatu");
  agenda.insereaza(1113,"Farna");
  agenda.insereaza(1114,"Bordu");
  agenda.insereaza(1115,"Ionescu");
  agenda.insereaza(1116,"Popescu");
  agenda.insereaza(1117,"Ion");
  agenda.insereaza(1118,"Vasile");
  agenda.insereaza(1119,"Milas");
  agenda.insereaza(6060,"Mocanu");
  agenda.insereaza(2524,"Vintila");
  agenda.insereaza(2610,"Zeda");
  agenda.insereaza(4753,"Istrate");
  agenda.insereaza(7421,"Panait");
  agenda.insereaza(3112,"Ghergu");
  agenda.insereaza(9428,"Pana");
  agenda.insereaza(9694,"Dinu");
  agenda.insereaza(4743,"Burca");
  agenda.insereaza(8231,"Dumitru");

// Urmeaza incercari de inserare a unor elemente deja existente.
// Ele nu vor mai fi efectuate.

  agenda.insereaza(9428,"Pana");
  agenda.insereaza(9694,"Dinu");
  agenda.insereaza(4743,"Burca");
  agenda.insereaza(8231,"Dumitru");

// se efectueaza 10 cautari de numere citite de la tastatura
 for(int i=0;i<10;i++)
 {
    long nr;
    elem * tmp;
    cout<<"Tasteaza un numar de telefon : ";
    cin>>nr;
```

```
    if((tmp=agenda.cauta_numar(nr))!=NULL)
        cout<<"Am gasit : "<<nr<<" "<<
            tmp->nume<<nl<<nl;
    else cout<<"Numarul "<<nr<<" nu exista in
              agenda !"<<nl<<nl;
  }
}
```

3.11.3. Mecanism perfecţionat de gestiune a memoriei

După cum am mai subliniat, unul din aspectele limbajului C++ ce suportă îmbunătăţiri seriose din punct de vedere a vitezei de execuţie, este mecanismul de gestiune al memoriei. În cartea "The C++ Programming Language" ([STRO] p.162-164), Bjarne Stroustrup dă un exemplu de îmbunătăţire a performanţelor mecanismului standard. Atît doar că acel program este prezentat ca o aplicaţie particulară, legată de o clasă anume. Graţie tehnicii de lucru cu "PROJECT"-uri, vom generaliza exemplul lui Stroustrup, aducîndu-i şi sensibile îmbunătăţiri calitative. Programul de mai jos poate fi utilizat cu succes şi în practică, generalitatea lui făcîndu-l util în orice situaţie de folosire intensivă a alocării dinamice de memorie. Ca performanţe atinse de acesta, putem preciza că, faţă de mecanismul implicit, aduce o scădere a timpului consumat cu alocări/eliberări de memorie de circa 19-20%. În cazul unei liste simplu înlănţuite cu 10 elemente cîştigul nu este deloc evident, dar la aplicaţii cu numeroase operaţii de alocare/eliberare cîştigul devine cu adevărat substanţial.

Să vedem care ar fi principiile de bază, după care să analizăm programul propus. Ideea novatoare constă în a aloca SIMULTAN un număr mai mare de entităţi (100 - 1000, în funcţie de aplicaţie), care să fie gestionate apoi de un mecanism propriu. Ce se înţelege prin "gestionate"? În principiu este vorba de:

a. Un alocator: el alocă la un moment dat o listă de N entităţi. De cîte ori programul aplicativ are nevoie de o nouă entitate, se va returna un pointer la următorul element disponibil din listă. În momentul în care lista se goleşte, se va aloca memorie pentru o nouă listă, tot de N elemente.
b. Un eliberator: în loc să elibereze memoria folosită de o entitate, obiectul în cauză se va adăuga ca nou element în capul listei de entităţi disponibile (gestionată de alocator). În acest mod se vor evita secvenţe de operaţii "new" şi "delete".

Un alt avantaj al metodei propuse constă în modul de organizare a clasei "inventar". Un obiect al acestei clase nu este altceva decît un pointer la o listă simplu înlănţuită. Care ar fi inovaţia adusă de noi? În cadrul clasei se va

introduce un membru static ce va contoriza numărul de instanţieri (în cazul nostru numărul de liste simplu înlănţuite). Astfel, nu va fi necesară eliberarea efectivă a tuturor entităţilor alocate DECÎT în momentul în care nu mai există nici o instanţiere a respectivei clase. Şi asta pentru că atîta vreme cît mai există obiecte de acest tip, memoria utilizată de instanţierile distruse nu trebuie eliberată, ci poate fi adaugată în lista de elemente disponibile.

Ca dezavantaj al metodei am putea putea aminti consumul mai ridicat de memorie. Dar, cum dezideratul constă în creşterea vitezei, era de presupus că ea se va face pe seama altui parametru de funcţionare a programului.

Ca temă, propunem o analiză serioasă a genurilor de aplicaţii în care un astfel de gestionar de memorie este eficient (atît ca timp de execuţie, cît şi ca memorie consumată).

În cele ce urmează, vom prezenta pe rînd fişierele programului:

a) **ALOCA_M.H**

```
// contine structurile de date folosite de catre gestionarul de memorie

struct LISTA {
 OBIECT * ob;        // "OBIECT" e definit in "date.h"
 LISTA  * next;
};

class gestionar_memorie {
 LISTA * pLista;
 LISTA * pRecuperate;

public :
 gestionar_memorie();
 ~gestionar_memorie();
 OBIECT * aloca_memorie();
 void elibereaza_memorie(OBIECT *);
};
```

b) **DATE.H**

```
// este folosit ca "include" in "GEST_MEM.CPP"

# include "inventar.h"

// "elem" va fi definit in "inventar.h"
# define OBIECT elem
```

```
# define NUMAR_OBIECTE 4
```

c) GEST_MEM.CPP

```
// contine definitiile metodelor clasei "gestionar_memorie"

# include "date.h"
# include "aloca_m.h"

# include <stdio.h>
# include <process.h>

// Nu exista inca cele doua liste
gestionar_memorie::gestionar_memorie()
{
 pLista=NULL;
 pRecuperate=NULL;
}

gestionar_memorie::~gestionar_memorie()
{
 LISTA * tmp;

 while(pLista!=NULL) {
    delete pLista->ob;
    tmp=pLista;
    pLista=pLista->next;
    delete tmp;
 }

// lista data de "pRecuperate" nu mai trebuie eliberata
// deoarece in acest moment este vida
}

// Daca mai exista obiecte disponibile, va returna un pointer la primul
// dintre ele; in caz contrar face acelasi luc ru,dar alocind in prealabil
// memorie
OBIECT * gestionar_memorie::aloca_memorie()
{
 LISTA * valDeReturnat=pLista;

 if(valDeReturnat==NULL)
// nu exista memorie disponibila; se va aloca memorie pentru un nou "sir"
```



```
{
   LISTA * temporar=(LISTA *)
          new char[NUMAR_OBIECTE*sizeof(LISTA)];
   OBIECT * tmp1=(OBIECT *)
          new char[NUMAR_OBIECTE*sizeof(OBIECT)];

// Se testeaza si "temporar" si "tmp1" deoarece s-ar putea ca
// lungimea lui "OBIECT" sa fie mai mica decit a lui "LISTA";
// in astfel de situatii exista posibilitatea (redusa, ce-i drept)
// ca sa fie suficienta memorie pentru a-crea un "OBIECT", nu
// insa si o "LISTA", si atunci "temporar" va fi NULL, iar "tmp1" nu.
// In toate celelalte situatii ar fi suficient de testat doar "tmp1".

    if( temporar==NULL || tmp1==NULL ) {
       printf("MEMORIE INSUFICIENTA\n");
       exit(1);
    }

// urmeaza ca fiecarui element din sirul "temporar" sa i se
// ataseze memoria necesara cite unui "OBIECT" ( NU UITATI :
// "LISTA" e o lista simplu inlantuita de pointeri la instantieri ale clasei
// "OBIECT" )

    valDeReturnat=temporar++;
    valDeReturnat->next=temporar;
    valDeReturnat->ob=tmp1++;

    for(int i=1;i<NUMAR_OBIECTE-1;i++,tmp1++){
       temporar->ob=tmp1;
       temporar->next=temporar+1;
       temporar=temporar->next;
     } // SFIRSIT "FOR"

    temporar->next=NULL;
    temporar->ob=tmp1;

  } // SFIRSIT"IF"

// primul element al listei initializate se va folosi imediat; de aceea el
// va fi pus chiar acum in "capul" listei cu elemente "de recuperat"

  pLista=valDeReturnat->next;
  valDeReturnat->next=pRecuperate;
```

```
   pRecuperate=valDeReturnat;
   return valDeReturnat->ob;
}

// Va asocia memoria folosita de obiectul disponibilizat unui element din
// lista "de recuperat", pe care-l va elimina din aceasta lista si-l va adauga
// la cea cu obiecte disponibile (in loc sa se elibereze memoria nefolosita,
// ea va fi pusa cu prima ocazie la dispozitia utilizatorului
void gestionar_memorie::elibereaza_memorie(OBIECT*
                                                pObiect)
{
 LISTA * tmp=pRecuperate;

// eliminare din lista "de recuperat"
 pRecuperate=pRecuperate-next;

// reinitializare si adaugare in lista cu elemente disponibile
 tmp->ob=pObiect;
 tmp->next=pLista;
 pLista=tmp;
}
```

d) INVENTAR.H

```
// descrie structurile de date folosite in clasa "inventar"

struct elem {
 char nume_obiect[20];
 int  numar_inventar;
 elem * urmator;
};

class gestionar_memorie;

class inventar {
    elem * cap_lista;

    static gestionar_memorie* gm;
    static int nr_obiecte;

    void distruge_inventar(void);
    void tipareste(void);
```



```
public :
    inventar(void);
    ~inventar(void);
    void adauga_in_lista(char *,int);
    void elimina_din_lista(int);
};
```

e) INVENTAR.CPP

```
// contine definitiile metodelor clasei "inventar"

# include <iostream.h>
# include <process.h>
# include <string.h>
# include <conio.h>

# include "date.h"
# include "aloca_m.h"

# define nl "\n"

# define DA 1
# define NU 0

int inventar::nr_obiecte=0;
gestionar_memorie*
          inventar::gm=new gestionar_memorie;

//Initializari
inventar::inventar(void)
{
 cap_lista=NULL;

 if(gm==NULL) {
    cout<<"MEMORIE INSUFIECIENTA"<<nl;
    exit(1);
 }
 nr_obiecte++;
}

inventar::~inventar(void)
{
 nr_obiecte--;
```

```
 distruge_inventar();
 if(nr_obiecte==0) {
     delete gm;
     cout<<"GATA\n";
 }
}

void inventar::distruge_inventar(void)
{
 while(cap_lista!=NULL)
     elimina_din_lista(cap_lista->numar_inventar);
}

// Atentie: in metoda "adauga_in_lista" nu se face validarea noului
// introdus (nu se verifica unicitatea numarului de inventar) !!!

void inventar::adauga_in_lista(char* nm,int nr)
{
 elem * tmp=gm->aloca_memorie();
 strcpy(tmp->nume_obiect,nm);
 tmp->numar_inventar=nr;
 tmp->urmator=cap_lista;
 cap_lista=tmp;

 tipareste();
}

// Tipareste intreaga lista de elemente
void inventar::tipareste()
{

 elem * tmp=cap_lista;

 cout<<nl<<nl<<"INVENTARUL COMPLET :"<<nl;

 while(tmp!=NULL) {
     cout<<tmp->nume_obiect<<" "<<
         tmp->numar_inventar<<nl;
     tmp=tmp->urmator;
  }
 getch();
}
```



```
// Elimina din lista un obiect identificat prin numarul de inventar
void inventar::elimina_din_lista(int numar)
{
 elem *, tmp=cap_lista,* pDeEliberat;
 int ExistaElement=NU;

 if(cap_lista!=NULL)
     if(cap_lista->numar_inventar==numar) {
        // eliminare cap de lista
         ExistaElement=DA;
         pDeEliberat=cap_lista;
         cap_lista=cap_lista->urmator;
     }
     else
         while(tmp->urmator!=NULL &&
               ExistaElement==NU ) {

             // se cauta secvential elementul dorit
             if(tmp->urmator->numar_inventar==numar){
                 ExistaElement=DA;
                 pDeEliberat=tmp->urmator;
                 tmp->urmator=tmp->urmator->urmator;
             }
             else tmp=tmp->urmator;
         }

 if(ExistaElement==DA) {
     gm->elibereaza_memorie(pDeEliberat);
     tipareste();
 }
}
```

f) M_PRJ.CPP

```
// Programul principal

# include <iostream.h>
# include <process.h>

# include "inventar.h"

void main(void){
 inventar i1,i2, * i3;
```

```
i3=new inventar;

if(i3==NULL)
{
    cout<<"\n\n\tMEMORIE INSUFICIENTA !!!";
    exit(1);
}
cout<<"ADAUGARE :\n";
for(int i=0;i<1000;i++) {
    i1.adauga_in_lista("qwerty",i);
}

cout<<"ELIMINARE :\n";
for(i=0;i<1000;i++) {
    i1.elimina_din_lista(1000-i);
}

cout<<"ADAUGARE :\n";
for(i=0;i<1000;i++) {
    i2.adauga_in_lista("qwerty",i);
}

cout<<"ELIMINARE :\n";
for(i=0;i<1000;i++) {
    i2.elimina_din_lista(1000-i);
}

cout<<"ADAUGARE :\n";
for(i=0;i<1000;i++) {
    i3->adauga_in_lista("qwerty",i);
}

cout<<"ELIMINARE :\n";
for(i=0;i<1000;i++) {
    i3->elimina_din_lista(1000-i);
}

delete i3;
}
```


4. MOŞTENIREA

4.1. Introducere

Vom începe explicarea conceptului de MOŞTENIRE printr-un exemplu legat de familia de funcţii "SCANF" ("scanf", "fscanf" şi "sscanf"). Într-o scurtă paranteză, vom reaminti cîteva caracteristici ale lor:

- **int scanf**(Format,Arg1,Arg2,...,ArgN)

  Citeşte de la dispozitivul standard de intrare "N" valori, în conformitate cu informaţiile din "Format".

- **int fscanf**(Pointer_La_Fisier,Format,Arg1,Arg2,...,ArgN)

  Citeşte din fişierul desemnat de "Pointer_La_Fisier" "N" valori, în conformitate cu informaţiile din "Format".

- **int sscanf**(Pointer_La_Buffer,Format,Arg1,Arg2,...,ArgN)

  Citeşte din bufferul desemnat de "Pointer_La_Buffer" "N"valori, în conformitate cu informaţiile din "Format".

După cum probabil aţi remarcat deja, modul de utilizare a celor trei funcţii este cvasi-identic, variind doar sursa din care vor fi citite valorile în cauză: dispozitivul standard de intrare, un fişier sau un buffer de caractere. O astfel de asemănare frapantă ne poate duce cu gîndul la faptul că, în fond, ar putea exista foarte bine o funcţie generală "_SCANF_", ale cărei proprietăţi sînt "moştenite" de către cele trei funcţii de bibliotecă. Şi, pentru că fiecare dintre procedurile "moştenitoare" are o proprietate caracteristică (sursa de unde preia valorile), sîntem de fapt în faţa a trei tipuri de funcţii aparţinînd uneia şi aceleiaşi FAMILII.

Iată, deci, în cîteva cuvinte la ce se reduce întreg mecanismul de MOŞTENIRE. Avînd o CLASĂ oarecare "B", putem defini o altă CLASĂ "D" care să MOŞTENEASCĂ (sau, dacă preferaţi, să PREIA) TOATE caracteristicile primeia, la care să poată adăuga ALTELE NOI, proprii doar acesteia din urmă. Prima clasă se va numi "CLASA DE BAZĂ", iar cea de a doua "CLASA DERIVATĂ".

Conform specialiştilor firmei BORLAND, "mecanismul de moştenire este, poate, singura mare diferenţă între C şi C++" ([BOR1], p. 141). Cu riscul de a ne repeta, dorim să subliniem încă o dată un lucru deosebit de important: deşi MOŞTENIREA stă la baza altor concepte novatoare (cum ar fi polimorfismul), cheia de boltă a programării obiectuale rămîne ÎNCAPSULAREA. Pentru că, să nu uităm, în lipsa acesteia din urmă, MOŞTENIREA nu ar fi avut nici un suport practic.

4.2. Clase derivate

În cele ce urmează vom da un prim exemplu de derivare a unei clase "hard_disk_2" din clasa "floppy_disk":

Exemplul 1

```
enum stare_operatie { REUSIT , ESEC };
enum stare_protectie_la_scriere { PROTEJAT ,
                                  NEPROTEJAT };

class floppy_disk {

protected:

// cuvintul cheie "protected" permite declararea unor membrii ne-pu-
// blici, care sa poata fi accesati de catre eventualele clase derivate din
// "floppy_disk"

 stare_protectie_la_scriere indicator_protectie;
 int capacitate,nr_sectoare;
public :
 stare_operatie formatare(void);
 stare_operatie citeste_pista(int drive,
          int sector_de_start,
          int numar_sectoare,void * buffer);
 stare_operatie scrie_pista(int drive,
          int sector_de_start,
          int numar_sectoare,void * buffer);
 stare_operatie protejeaza_la_scriere(void);
};

// clasa "hard_disk_1" defineste un hard-disk FARA a profita de
// avantajele mostenirii

class hard_disk_1 {
 stare_protectie_la_scriere indicator_protectie;
 int capacitate,nr_sectoare;
 int  numar_partitii;
public :
 stare_operatie parcheaza_disc(void);
 stare_operatie formatare(void);
 stare_operatie citeste_pista(int drive,
       int sector_de_start,
```



```
   int numar_sectoare,void * buffer);
 stare_operatie scrie_pista(int drive,
   int sector_de_start,
   int numar_sectoare,void * buffer);
 stare_operatie protejeaza_la_scriere(void);
};

// clasa "hard_disk_2" defineste un hard-disk; se utilizeaza
// avantajele mostenirii proprietatilor clasei "floppy_disk"

class hard_disk_2 : public floppy_disk{
 int numar_partitii;
public :
 stare_operatie parcheaza_disc(void);
};

// din definirea metodelor de mai jos reies drepturile de
// accesare ale membrilor clasei derivate

stare_operatie hard_disk_2::parcheaza_disc(void)
{
// CORECT: accesare membru "protected"
 indicator_protectie=PROTEJAT;
// ...
 return REUSIT;
}

void functie(void)
{
 hard_disk_2 hd1;
 hd1.formatare(); // CORECT
 hd1.indicator_protectie=NEPROTEJAT;// EROARE:
                                    // incercare de
                                    // accesare a unui
                                    //  membru protejat
}
```

Incontestabil, cea de a doua metodă de a defini un hard-disk este mult mai elegantă, mai eficace şi mai uşor de înţeles. Să analizăm cu atenţie secvenţa de mai sus. Prin enunţul

```
class hard_disk_2 : public floppy_disk
```

se indică compilatorului următoarele informaţii:

- clasa "hard_disk_2" va fi DERIVATA din clasa "floppy_disk";
- toţi membrii de tip "public" ai clasei "floppy_disk" vor fi moşteniţi (şi deci vor putea fi folosiţi) ca "public" de către clasa "hard_disk_2";
- toţi membrii "protected" ai lui "floppy_disk" vor putea fi utilizaţi ca fiind "protected" în cadrul clasei "hard_disk_2".

Este evident că eficacitatea nu înseamnă doar faptul că în declaraţia clasei derivate nu mai apar informaţiile moştenite (ele fiind automat luate în considerare de către compilator). În plus, nu mai trebuie rescrise nici măcar funcţiile-membru ale clasei de bază, ele putînd fi refolosite EXACT în maniera în care au fost definite. Dar, aşa cum vom vedea în secţiunile viitoare, acest lucru nu înseamnă că, dacă dorim, nu putem să le redefinim, găsindu-le o cu totul altă funcţionalitate.

4.3. Drepturi de accesare a membrilor "moşteniţi"

În secţiunea "3.2." ofeream o sintaxă simplificată a declarării unei clase:

```
specificator_clasa nume_clasa {
 [ [ private : ] lista_membri_1 ]
 [ [ public  : ] lista_membri_2 ]
};
```

De data aceasta vom rafina acea formă, şi vom obţine:

```
specificator_clasa nume_clasa_derivata
[ :     [ modificator_acces_1 ] nume_clasa_baza_1
 [ , [ modificator_acces_2 ] nume_clasa_baza_2 ]
    [ , ... ]
 ]
]

{
 [ [ private   : ] lista_membri_1 ]
 [ [ protected : ] lista_membri_2 ]
 [ [ public    : ] lista_membri_3 ]
} [lista_obiecte];
```

Pentru început, ne vom ocupa de cazul în care există un singur modificator de acces ("modificator_acces_1") şi o singură clasă de bază ("nume_clasa_baza_1"). În secţiunea "4.6." vom trata cazul general (mai mulţi modificatori de acces şi clase de bază), numit şi MOŞTENIRE MULTIPLĂ.

În sintaxa prezentată există trei noutăţi:

- exprimarea relaţiei de "MOŞTENIRE" ("nume_clasa_derivata" fiind DERIVAT din "nume_clasa_baza_1");
- prezenţa lui "modificator_acces";
- apariţia secţiunii "protected".

În funcţie de tipul lui "specificator_clasa" ("class" sau "struct"), "modificator_acces" va lua valoarea IMPLICITĂ "private" sau "public". Rolul acestui "modificator_acces" este ca împreună cu specificatorii "public", "private" sau "protected" întîlniţi în cadrul declarării clasei de bază, SĂ HOTĂRASCĂ dreptul de accesare a membrilor moşteniţi de către o clasă derivată. ATENŢIE: modificatorul de acces NU MODIFICĂ ÎN NICI UN FEL dreptul de accesare a membrilor clasei de bază!

Nu am amintit nimic despre "union"-uri, deoarece ele NU POT FI NICI CLASE DE BAZĂ, NICI CLASE DERIVATE!!!

În continuare, vom prezenta un tabel care sintetizează drepturile de accesare a membrilor unei clase derivate "D" în funcţie de drepturile de accesare a membrilor clasei de bază "B" şi valoarea lui "modificator_acces".

Drept de acces în clasa de bază	**Modificator de acces**	**Drept de acces în clasa derivată**
PUBLIC *PRIVATE* *PROTECTED*	*PUBLIC* *PUBLIC* *PUBLIC*	*PUBLIC* *inaccesibil* *PROTECTED*
PUBLIC *PRIVATE* *PROTECTED*	*PRIVATE* *PRIVATE* *PRIVATE*	*PRIVATE* *inaccesibil* *PRIVATE*

Vom începe explicarea acestor relaţii prin două observaţii legate de cuvîntul cheie "protected":

- el NU poate fi folosit ca "modificator_acces" în cadrul unei relaţii de moştenire;
- rolul său se limitează la a permite accesarea în cadrul unei clase derivate a unor membrii ne-publici ai clasei de bază.

Continuăm comentarea tabelului de mai sus cu o altă remarcă. Indiferent de modificatorul de acces folosit, MEMBRII PRIVAŢI AI UNEI CLASE DE BAZĂ NU VOR PUTEA FI ACCESAŢI DIRECT ÎN CLASA DE-

RIVATĂ! Este o restricţie foarte logică, în lipsa căreia s-ar încălca însăşi ideea de dată privată (adică accesibilă NUMAI metodelor respectivei clase). Dar, NOTA BENE, în ciuda imposibilităţii de a-i accesa direct, ei continuă să EXISTE în cadrul fiecărei instanţieri a clasei derivate. La momentul potrivit, vom vedea că ei pot fi utilizaţi PRIN INTERMEDIUL METODELOR MOŞTENITE de la clasa de bază (vezi exemplul din secţiunea "4.7.").

Şi încă o observaţie: orice "FRIEND" al unei clase derivate are exact aceleaşi drepturi şi posibilităţi de a accesa membrii clasei de bază ca orice alt membru al clasei derivate.

Restul informaţiilor conţinute în tabel nu vor fi comentate, ci exemplificate în cele ce urmează:

Exemplul 1

```
class baza {
 private :
     int a;
     int f1(void);
 protected :
     int b;
     int f2(void);
 public :
     int c;
     int f3(void);
};

class derivata_1 : public baza {
     int a1;
 protected :
     int a2;
 public :
     int a3;
     int fd1(void);
};

class derivata_2 : private baza {
     int a1;
 protected :
     int a2;
 public :
     int a3;
     int fd2(void);
};
```



```
int baza::f3(void)
{
 a=b=c=0; // CORECT: "f3" poate accesa orice membru al clasei
          // "baza"
 return a;
}

// OBSERVATIE:
// In cadrul celor doua metode descrise mai jos ("fd1" si "fd2"),
// DREPTURILE DE ACCESARE a membrilor clasei de baza sint
// IDENTICE. Diferentele apar doar in momentul accesarii
// acestor membri in cadrul unor functii ne-membru.

int derivata_1::fd1(void)
{
 a1=a2=a3=0; // CORECT

 b=f3();    // CORECT: "b" si "f3" sint mostenite cu drept de acces

 c=f2();    // CORECT: "c" si "f2" sint mostenite cu drept de acces

 a=f1();    // EROARE: atit "a" cit si "f1()" sint mostenite,
            // dar nu pot fi accesate in clasa derivata

  return b+c; // CORECT
}

int derivata_2::fd2(void)
{
 a1=a2=a3=0; // CORECT

 b=f3();    // CORECT: "b" si "f3" sint mostenite cu drept de acces

 c=f2();    // CORECT: "c" si "f2" sint mostenite cudrept de acces

 a=f1();      // EROARE: atit "a" cit si "f1()" sint mostenite,
              // dar nu pot fi accesate in clasa derivata

 return b*f2()+c*f3(); // CORECT
}
```

```
void functie(void)
{
 baza        ob_baza;
 derivata_1 ob_der1;
 derivata_2 ob_der2;

 ob_baza.c=ob_baza.a-ob_baza.b;
// EROARE: doar "ob_baza.c" este accesibil in functii ne-membru

 ob_der1.a=ob_der1.b*ob_der1.c/ob_der1.f1()
          +ob_der1.f2()-ob_der1.f3();
 // EROARE: doar "c" si "f3" sint accesibile in functii ne-membru

 ob_der2.b=ob_der2.f3()-ob_der2.a*ob_der2.c/
          ob_der2.f2()*ob_der2.f1();;
// EROARE: nici unul din membrii mosteniti ("a","b","c","f1","f2","f3")
// nu este accesibil in functii ne-membru (in cazul nostru "functie")
}
```

Deoarece o clasă derivată poate fi folosită la rîndul ei ca şi clasă de bază, este necesară o atenţie deosebită în alegerea celei mai bune combinaţii "tip membri"-"modificator acces". În caz contrar, există riscul ca eventuala clasă "nepot" să piardă mult din elasticitate.

O remarcă foarte interesantă ţine de istoricul C++: versiunea iniţială "AT&T 1.00" nu conţinea cuvîntul cheie "protected". Stroustrup argumenta imposibilitatea de a accesa direct dintr-o clasă derivată un membru ne-public al clasei de bază prin încălcarea statutului de membru privat (adică accesibil DOAR clasei ce-l poseda). Totuşi, în cele din urmă, necesitatea de a moşteni şi manipula membri ne-publici ai claselor de bază a dat naştere compromisului numit "protected". Din exemplul de mai jos reies cîteva caracteristici deosebit de importante în utilizarea claselor derivate.

Exemplul 2

```
class BAZA {
 // ...
public :
 int contor1,contor2;

 void f1(void) {
 // ...
 }

 void f2(void) {
 // ...
```



```
 }

 void f3(int a) {
 // ...
 }
};

class DERIVAT1 : public BAZA{
// ...
public :
 int contor1; // definirea unui nou membru, purtind acelasi
              // nume cu cel mostenit

// urmeaza redefinirea metodelor clasei de baza

 void f1(void) {
 // ...
    BAZA::f1();      // apelul metodei "f1" a clasei "BAZA"
 }

 int f2(void) {
 // ...
    BAZA::f2();      // apelul metodei "f2" a clasei "BAZA"
    return 0;
 }

 void f3(void)
 {
    BAZA::f3(10);    // apelul metodei "f3" a clasei "BAZA"
 }
};

class DERIVAT2 : BAZA{ // implicit, modificatorulde acces
                       // folosit este "private"
// ...
public :
 BAZA::contor1;
 BAZA::f1;     // cei doi membri mosteniti sint FORTATi sa
               // REDEVINA "public"
};

void main(void)
{
```

```
DERIVAT1 d1;
DERIVAT2 d2;

d1.f1();
d1.f2();            // apelul metodelor redefinite

d1.BAZA::f1();      // apelul metodei clasei "BAZA"
d1.contor1=0;       // initializarea noului membru
d1.BAZA::contor1=1;     // initializarea membrului mostenit

d2.f1();
d2.contor1=0;       // utilizarea membrilor mosteniti si
                    // FORTATI sa REDEVINA "public"

d2.f2();            // EROARE: membru privat inaccesibil
d2.contor2=1;       // EROARE: membru privat inaccesibil
}
```

În primul rînd vom analiza clasa "DERIVAT1". Se observă că, în cadrul acestei clase, vor fi redefinite ("OVERLOADED") toate cele trei metode ale clasei de bază. O facilitate de limbaj permite însă accesarea de către orice membru al clasei "DERIVAT1" a tuturor acestor metode aparţinînd clasei "BAZA" şi redefinite în "DERIVAT1". Modul de folosire este foarte simplu: funcţia în cauză va fi prefixată cu numele "clasei posesoare" (în cazul nostru "BAZA") urmat de "::". În lipsa prefixării cu "BAZA::", efectul va fi ori un apel recursiv infinit (în cazul metodelor "f1" si "f2"), ori o eroare la compilare (este cazul lui"f3", care e apelată cu un parametru întreg, deşi a fost redefinită ca neavînd nici un parametru).

Trebuie amintit încă un lucru: deşi în clasa "DERIVAT1" am adăugat încă o variabilă membru "contor1", în funcţia "main" pot fi accesate AMBELE variabile "contor1" (atît cea moştenită, cît şi cea adaugată). Diferenţierea lor se va face prin prefixarea variabilei moştenite cu "BAZA::".

Vom mai sublinia că prefixarea cu numele clasei de bază este valabilă şi în afara corpului metodelor (vezi cazurile instrucţiunilor "d1.BAZA::f1();" şi "d1.BAZA::contor1=1;").

A doua noutate este legată de clasa "DERIVAT2". Exprimarea, oarecum ciudată

```
public :
 BAZA::contor1;
 BAZA::f1;
```


nu ascunde nimic altceva decît posibilitatea de a forţa unii membrii să redevină de tip "public". Acest lucru va fi făcut ÎN CIUDA REGULILOR DE MODIFICARE A DREPTULUI DE ACCES din cadrul mecanismului de moştenire (deci nu este nicidecum vorba de adăugări de noi membrii). Aşa cum se vede, atît în cazul variabilelor cît şi în cel al metodelor, este suficientă simpla precizare a numelui şi a clasei de proveniență. De altfel, încercarea de a da mai multe amănunte despre respectivul membru (tipul variabilei, respectiv valoarea returnată sau tipul parametrilor) va fi semnalizată ca eroare de compilare. De remarcat faptul că, în acelaşi mod, se poate efectua şi forţarea membrilor ce şi-au pierdut calitatea de a fi "protected" (datorită unei moşteniri de tip privat). Există şi o restricţie: indiferent de modificatorul de acces utilizat, NU ESTE PERMISĂ transformarea unui membru PRIVAT al clasei de bază într-unul PUBLIC sau PROTECTED al celei derivate.

O altă chestiune ce trebuie discutată constă în posibilitatea de a se efectua conversii de tipul "B*"->"D*" şi "D*"->"B*", unde prin "D" am notat o clasă derivată din clasa "B". Exemplul de mai jos ne va lămuri pe deplin asupra problemei.

Exemplul 3

```
class baza{
 // ...
};

class derivat_1 : public baza{
 // ...
};

class derivat_2 : private baza{
 // ...
};

void main(void)
{
 baza      *  pb,ob;
 derivat_1 *  pd1,od1;
 derivat_2 *  pd2,od2;

 pb=&od1;     // CORECT: conversie implicita

 pb=&od2;     // EROARE: nu se poate face conversie implicita

 pd1=&ob;     // EROARE: nu se poate face conversie implicita
 pd2=&ob;     // EROARE: nu se poate face conversie implicita
```

```
    pb=(baza*) &od2; // CORECT: conversie explicita

    pd1=(derivat_1*) &ob; // CORECT: conversie explicita
    pd2=(derivat_2*) &ob; // CORECT: conversie explicita
}
```

Reluînd, în cuvinte, cele prezentate în cadrul exemplului, conversia "DERIVAT*"->"BAZA*" se poate face:

IMPLICIT: – dacă "DERIVAT" moşteneşte pe "BAZA" în manieră publică;

EXPLICIT: – dacă "DERIVAT" se obţine din "BAZA" prin moştenire în manieră privată.

Conversia "BAZA*"->"DERIVAT*" nu poate fi făcută DECÎT EXPLICIT, prin operatorul "cast".

Să analizăm următoarea situaţie: o variabilă publică a clasei "baza" devine privată în urma moştenirii ei de către clasa "derivat_2". Făcînd o conversie "derivat_2*"->"baza*", printr-un pointer la "baza" vom putea accesa respectiva variabilă ca şi cum ar fi publică. Cu toate că acest lucru este POSIBIL, nu este DELOC RECOMANDABIL. Înainte de a trece mai departe, vom relua avertismentul lui Bjarne Stroustrup: ÎNCĂLCAREA DREPTURILOR DE ACCES PRIN CONVERSII EXPLICITE între pointeri la tipuri de date legate între ele prin relaţia de moştenire riscă să ducă la erori aproape imposibil de detectat. De aceea, în astfel de situaţii, se recomandă o atenţie deosebită.

4.4. Constructori şi destructori

Avînd în vedere că scopul claselor derivate este acela de a permite adăugarea unor noi caracteristici unei clase de bază, este evident că, în general, clasele derivate vor avea nevoie de constructori mai complecşi decît cele de bază. Pentru a uşura cît mai mult munca programatorului şi pentru a avea o comportare cît mai naturală, constructorii claselor de bază vor fi apelaţi IMPLICIT, dacă acest lucru este posibil. În caz contrar, va fi necesară specificarea EXPLICITĂ a parametrilor în cauză (vezi exemplul de mai jos). Specificarea explicită a parametrilor constructorului se poate face şi dacă programatorul doreşte apelul unui constructor anume din setul de constructori ai respectivei clase. Deoarece o clasă derivată nu este altceva decît o construcţie "în jurul" unei clase de bază, este logic ca mai întîi să se apeleze constructorii claselor de bază, şi doar după aceea să se execute acţiunile specifice constructorului clasei derivate.

În cazul destructorilor, problema se va pune exact invers: mai întîi va fi apelat destructorul clasei derivate şi abia după aceea cel al clasei de bază.

Exemplul 1

```
# include <iostream.h>
# include <conio.h>

class BAZA
{
 protected :
    int cont;
 public :

    void SET_CONTOR(int val) { cont=val; }
    void TYPE_CONTOR(void) {
       cout<<"\n\n CONTOR = "<<cont;
    }

    BAZA(int a=0)
    {
       SET_CONTOR(a);
       cout << "\nConstructorul clasei BAZA;\t
             CONTOR="<<cont<<"\n";
    }

    ~BAZA(void)
    {
       cout << "\nDestructorul clasei BAZA;\t
           CONTOR="<<cont<<"\n";
    }
};

class DERIVAT : public BAZA
{
public :
 DERIVAT(void)     // se va apela IMPLICIT "BAZA(0)" !
 {
    cout << "Constructorul clasei DERIVAT\n";
 }

 DERIVAT(int b) : BAZA(b) // apel explicit al constructorului
                          // clasei de baza
 {
```

```
        cout << "Alt constructor al clasei DERIVAT\n";
    }

    void TYPE_CONTOR(void)
    {
        cout<<"\n\n Contorul clasei derivate este "
            <<cont;
    }

    ~DERIVAT(void)
    {
        cout << "\nDestructorul clasei DERIVAT;";
    }

};

void main(void)
{
 clrscr();

 BAZA    b1,b2=1;
 DERIVAT d1,d2(2),d3(3);

 BAZA    * pb1;
 DERIVAT * pd1;

 pb1=&b2;
 pd1=&d2;

 pb1->TYPE_CONTOR();
 pd1->TYPE_CONTOR();

 cout<<"\n";
}
```

Putem urmări ordinea executării procedurilor programului ghidîndu-ne după succesiunea mesajelor afişate pe ecran:

Constructorul clasei BAZA; CONTOR=0

Constructorul clasei BAZA; CONTOR=1

Constructorul clasei BAZA; CONTOR=0
Constructorul clasei DERIVAT

Constructorul clasei BAZA; CONTOR=2
Alt constructor al clasei DERIVAT

Constructorul clasei BAZA; CONTOR=3
Alt constructor al clasei DERIVAT

CONTOR = 1

Contorul clasei derivate este 2

Destructorul clasei DERIVAT;
Destructorul clasei BAZA; CONTOR=3

Destructorul clasei DERIVAT;
Destructorul clasei BAZA; CONTOR=2

Destructorul clasei DERIVAT;
Destructorul clasei BAZA; CONTOR=0

Destructorul clasei BAZA; CONTOR=1

Destructorul clasei BAZA; CONTOR=0

La al doilea constructor al clasei derivate se vede foarte bine modul în care se vor specifica parametrii necesari constructorului clasei de bază. De notat doar că exprimarea

```
DERIVAT(int b) : BAZA(b)
```

nu era necesară, ea putînd fi înlocuită de mai simpla

```
DERIVAT(int b) : (b)
```

În acest moment trebuie să atragem atenţia asupra unui aspect prezentat în exemplul de mai jos.

Exemplul 2

```
class B
{
  B();      // EROARE: constructor inaccesibil clasei D
 ~B();      // EROARE: destructor inaccesibil clasei D
};

class D : B
{
};
```

Analizînd cele două structuri de date, tragem următoarea concluzie: ESTE INTERZIS CA ÎN CADRUL UNEI CLASE BAZA SĂ AVEM TOŢI CONSTRUCTORII SAU DESTRUCTORII PRIVAŢI. Avînd în vedere gravitatea unor astfel de situaţii (în care clasa derivată NU POATE ACCESA CONSTRUCTORUL SAU DESTRUCTORUL clasei de bază), erorile de acest gen vor fi semnalizate în faza de compilare.

Deşi pare fără sens, existenţa unor constructori şi destructori privaţi nu este lipsită de interes. Nu uitaţi, aceste metode vor putea fi folosite - de exemplu - în cadrul unor funcţii "friend".

Cu titlu de observaţie, subliniem că destructorii şi constructorii claselor de bază NU SE MOŞTENESC. Cu toate acestea, accesibilitatea lor este esenţială în procesul creării şi distrugerii instanţierilor clasei derivate.

Încheiem această secţiune prin analiza creării dinamice de obiecte fără a utiliza mecanismul implicit de gestionare a memoriei. Pentru înţelegerea exemplului ce urmează trebuie să mai facem o subliniere. Dacă în cadrul constructorului unei clase derivate se face o atribuire de genul "this=...", constructorul clasei de bază nu va fi apelat pînă în momentul în care atribuirea a fost făcută. Lucrurile se petrec astfel deoarece valoarea lui "THIS" în cadrul constructorului clasei de bază va fi una şi aceeaşi cu cea atribuită lui în constructorul clasei derivate. Reciproc, dacă în cadrul constructorului clasei de bază i se atribuie lui "THIS" o valoare, ea va fi folosită în continuare şi în constructorul clasei derivate!!!

Exemplul 3

```
# include <iostream.h>
# include <process.h>

class BAZA {
public :

 BAZA(void * pointer=NULL)
 {
    cout<<"\n\t\t La apelul constructorului BAZA()
         this="<<this;

    if(this==NULL)
       if(pointer!=NULL)
          this=(BAZA *) pointer;
       else
       {
          cout<<"\nEROARE : in \"BAZA()\" se
              forteaza this=NULL";
```



```
          exit(1);
       }
    else this=this; // Nu este inutil: vezi observatia de
                    // la sfirsitul exemplului !

    cout<<"\n\t\t La iesirea din BAZA() this="
        <<this;
 }
};

class DERIVAT : public BAZA{
public :

 DERIVAT(void * pointer=NULL)
 {
    cout<<"\n\t\t La apelul constructorului
        DERIVAT() this="<<this;

    if(this==NULL)
       if(pointer!=NULL)
          this=(DERIVAT *) pointer;
       else
       {
          cout<<"\nEROARE: in \"DERIVAT()\" se
              forteaza this=NULL";
          exit(1);
       }
    else this=this; // Nu este inutil: vezi observatia de
                    // la sfirsitul exemplului !

    cout<<"\n\t\t La iesirea din DERIVAT() this="
        <<this<<"\n\n";
 }
};

void main(void)
{
 cout<<"OBIECTE CREATE FARA ALOCARE DINAMICA DE
      MEMORIE :\n\n";
 cout<"\tBAZA";
 BAZA b1;
 cout<<"\n\tDERIVAT\n";
 DERIVAT d1;
```

```
cout<<"OBIECTE CREATE PRIN ALOCARE DINAMICA DE
     MEMORIE :\n\n";
cout<<"\tBAZA";
new BAZA(&b1);

cout<<"\n\tDERIVAT\n";
new DERIVAT(&d1);

// Fiecare din cele patru creari dinamice de obiecte ce apar in continuare
// ar provoca oprirea executiei programului. Acest fapt s-ar datora
// atingerii apelurilor functiei "exit" din cadrul celor doi constructori
// ("BAZA" si "DERIVAT").

// new BAZA(NULL);
// new BAZA();
// new DERIVAT(NULL);
// new DERIVAT();

}
```

În urma execuţiei acestui program, pe ecran se vor tipări următoarele mesaje:

OBIECTE CREATE FARA ALOCARE DINAMICA DE MEMORIE :

BAZA

La apelul constructorului BAZA() this=8FD3:0FFE
La iesirea din BAZA() this=8FD3:0FFE

DERIVAT

La apelul constructorului DERIVAT() this=8FD3:0FFC
La apelul constructorului BAZA() this=8FD3:0FFC
La iesirea din BAZA() this=8FD3:0FFC
La iesirea din DERIVAT() this=8FD3:0FFC

OBIECTE CREATE PRIN ALOCARE DINAMICA DE MEMORIE :

BAZA

La apelul constructorului BAZA() this=0000:0000
La iesirea din BAZA() this=8FD3:0FFE

DERIVAT

La apelul constructorului DERIVAT() this=0000:0000
La apelul constructorului BAZA() this=8FD3:0FFC
La iesirea din BAZA() this=8FD3:0FFC
La iesirea din DERIVAT() this=8FD3:0FFC

Ceea ce merită într-adevăr analizat în acest exemplu este constructorul clasei DERIVAT. Se observă că "IF"-ul care testează valoarea lui "this" are - oarecum surprinzător - două ramuri. Pe una se face atribuirea necesară în caz că "THIS" este NULL, iar pe cealaltă se va face o atribuire de tipul

```
this=this;
```

Deşi nu are nici un scop aparent, această atribuire este FUNDAMENTALĂ în cazul în care constructorul poate fi apelat atît cu "THIS!=NULL", cît şi cu "THIS==NULL". Vom justifica afirmaţia prin faptul că, dacă în cadrul constructorului clasei derivate se va detecta (LA COMPILARE) o atribuire către "THIS", apelul constructorului clasei de bază se va face DOAR DUPĂ EFECTUAREA ATRIBUIRII. Ei bine, de aici apare şi problema. S-ar putea ca atribuirea unei valori către "this" să se facă DOAR PE O SINGURĂ "RAMURĂ" A PROGRAMULUI (de exemplu, într-un IF se face doar pe "THEN", nu şi pe "ELSE"). În cazul în care execuţia va avea loc pe o altă ramură, apelul constructorului clasei de bază nu se va mai executa DEOARECE NU S-A ATINS INSTRUCŢIUNEA CONŢINÎND ATRIBUIREA CĂTRE "THIS". Iată deci o problemă cu totul neaşteptată, şi care poate da multă bătaie de cap programatorului neatent!

4.5. Metode virtuale. Utilizarea listelor eterogene

Prin definiţie, un tablou omogen este un tablou conţinînd elemente de acelaşi tip (de exemplu, un şir de întregi). După cum am spus, un pointer la o clasă "B" poate păstra adresa oricărei instanţieri a vreunei clase derivate din "B". Deci, avînd un şir de pointeri la obiecte de tip "B", înseamnă că, de fapt, putem lucra şi cu tablouri neomogene. Astfel de tablouri neomogene se vor numi ETEROGENE. Una din caracteristicile limbajului C++ constă tocmai în faptul că MECANISMUL FUNCŢIILOR VIRTUALE PERMITE TRATAREA UNIFORMĂ (sau, dacă preferaţi, ÎN ACEEAŞI MANIERĂ) a tuturor elementelor unui masiv de date ETEROGENE! Acest lucru este posibil datorită unor facilităţi ce ţin de asocieri făcute DOAR ÎN MOMENTUL EXECUŢIEI PROGRAMULUI, nu în momentul compilării!

Vom face o scurtă paranteză pentru a explica mai pe larg aceste principii generale. Să presupem că avem o clasă "B" care posedă o metodă publică "M". Din clasa "B" vom deriva mai multe clase "D1", "D2", "D3",..., "Dn". Dacă clasele

```
    {
        pB=p; t=tp;
    }

    void tipareste_valoare(void)
    {
        if( t== _BAZA_ )
          pB->tipareste_valoare();
        else
          if( t== _DERIVAT_1 )
             ((DERIVAT_1 *) pB)->
                      tipareste_valoare();
          else
             ((DERIVAT_2 *) pB)->
                      tipareste_valoare();
    }
};

void main(void)
{
 BAZA     a[2];
 DERIVAT_1 b1;
 DERIVAT_2 b2;

 LISTA_ETEROGENA  p[4];

 a[0].set_val(1);
 a[1].set_val(2);
 b1.set_val(3);
 b2.set_val(4);

 p[0].set(&a[0]);

 p[1].set(&b1,_DERIVAT_1);// se face o conversie IMPLI-
                          // CITA "DERIVAT_1"-> "BAZA*"

 p[2].set((BAZA *) &b2,_DERIVAT_2); // se face o conver-
                 // sie EXPLICITA "DERIVAT_2"-> "BAZA*"

 p[3].set(&a[1]);

 for(int i=0;i<4;i++)
    p[i].tipareste_valoare();
```



```
}
```

În urma execuţiei programului, pe ecran se vor afişa următoarele mesaje:

```
Element BAZA cu VALOARE = 1
Element DERIVAT_1 cu VALOARE = 3
Element DERIVAT_2 cu VALOARE = 4
Element BAZA cu VALOARE = 2
```

Pentru a reuşi o tratare uniformă a celor trei tipuri de obiecte a fost necesară crearea unei a patra clase, numită LISTA_ETEROGENA. Aceasta va păstra atît pointer-ul la obiectul respectiv, cît şi o informaţie suplimentară, referitoare la tipul obiectului referit de pointer. Tipărirea informaţiilor semnificative ale fiecărui obiect se va face în metoda "LISTA_ETEROGENA::tipareste_valoarea". În această funcţie membru sînt necesare o serie de teste pentru apelul metodei corespunzînd fiecărui tip de obiect. Vă daţi seama că, dacă în loc de 3 tipuri de obiecte am lucra cu 10 sau 100, soluţia ar deveni mult mai greoaie şi ne-elegantă.

Reamintim că în instrucţiunea

```
p[2].set((BAZA *) &b2,_DERIVAT_2);
```

operatorul "CAST" este absolut necesar. Conform celor afirmate în secţiunea "4.3.", dacă dintr-o clasă "B" derivăm în maniera privată o altă clasă "D", conversia "D*"->"B*" nu se poate realiza decît de maniera EXPLICITĂ.

În Exemplul 2 vom vedea o modalitate mult mai simplă şi elegantă de a trata OMOGEN un masiv de date eterogene. În Exemplul 1 tratarea omogenă se referea la instrucţiunea "for", în care, indiferent de tipul elementului curent, se va apela una şi aceeaşi funcţie-acţiune. Se vede foarte bine că, în acest caz, este vorba de o pseudo tratare uniformă, deoarece decizia de tratare diferenţiată este aminată de la nivelul lui "for" la cel al metodei "LISTA_ETEROGENA::tipareste_valoarea". Iată acum o rezolvare tipic obiectuală a problemei.

Exemplul 2

```
# include <iostream.h>

class BAZA {
 protected :
    int valoare;
 public :
    void set_val(int a) { valoare=a; }

    virtual void tipareste_valoare(void)
    {
```

```
        cout<<"Element BAZA cu VALOARE = "
            <<valoare<<"\n";
      }
};

class DERIVAT_1 : public BAZA {
 public :
    void tipareste_valoare(void)
    {
        cout<<"Element DERIVAT_1 cu VALOARE = "
            <<valoare<<"\n";
    }
};

class DERIVAT_2 : BAZA {
 public :
    BAZA::set_val;

    void tipareste_valoare(void)
    {
        cout<<"Element DERIVAT_2 cu VALOARE = "
            <<valoare<<"\n";
    }
};

class LISTA_ETEROGENA {
    BAZA * pB;
 public:
    void set(BAZA * p) { pB=p; }

    void tipareste_valoare(void)
    {
       pB->tipareste_valoare();
    }
};

void main(void)
{
 BAZA    a[2];
 DERIVAT_1 b1;
 DERIVAT_2 b2;

 LISTA_ETEROGENA p[4];
```



```
 a[0].set_val(1);
 a[1].set_val(2);
 b1.set_val(3);
 b2.set_val(4);

 p[0].set(&a[0]);
 p[1].set(&b1);
 p[2].set((BAZA *) &b2);
 p[3].set(&a[1]);

 for(int i=0;i<4;i++)
    p[i].tipareste_valoare();
}
```

Şi de astă dată, pe ecran vor fi afişate aceleaşi informaţii ca şi în cazul exemplului precedent:

Element BAZA cu VALOARE = 1
Element DERIVAT_1 cu VALOARE = 3
Element DERIVAT_2 cu VALOARE = 4
Element BAZA cu VALOARE = 2

Marea diferenţă între cele două exemple constă în aceea că, acum, clasa LISTA_ETEROGENA are o metodă "tipareste-valoarea" mult simplificată. În noua funcţie membru nu mai este necesară nici testarea tipului de pointer memorat şi nici conversia pointerului memorat la tipul original. Şi toate acestea doar din cauza prefixării metodei "BAZA::tipareste_valoarea" cu cuvîntul cheie "virtual". Practic, în urma întîlnirii lui "virtual", compilatorul va lua AUTOMAT şi FĂRĂ NICI UN ALT AJUTOR DIN PARTEA PROGRAMATORULUI toate deciziile necesare pentru efectuarea unui "LATE BINDING" corect.

Să încercăm să înţelegem modul de realizare al unui mecanism atît de puternic. Prefixarea unei metode "F" a clasei "B" cu cuvîntul cheie "virtual" are următorul efect: toate clasele ce o moştenesc pe "B" şi nu REDEFINESC funcţia "F", o vor moşteni întocmai. Să presupunem acum că o clasă "D", derivată din "B", va REDEFINI metoda "F". În această situaţie, COMPILATORUL VA AVEA SARCINA DE A ASOCIA UN SET DE INFORMAŢII SUPLIMENTARE, CU AJUTORUL CĂRORA SE VA PUTEA DECIDE (IN MOMENTUL EXECUŢIEI) CARE METODĂ "F" VA FI APELATĂ.

Observaţii

a. Deoarece funcţiile virtuale necesită memorarea unor informaţii suplimentare, instanţierile claselor care au metode virtuale vor ocupa în memorie mai mult loc decît ar fi necesar în cazul în care nu ar exista decît metode

ne-virtuale.

Trebuie amintite şi două restricţii: nici FUNCŢIILE NE-MEMBRU şi nici METODELE STATICE NU POT FI DECLARATE VIRTUALE!

b. Pentru ca mecanismul metodelor virtuale să poată funcţiona, metoda în cauză NU POATE FI REDECLARATĂ în cadrul claselor derivate ca având aceiaşi parametrii şi returnînd un ALT TIP de dată. În schimb, este permisă REDECLARAREA cu alt set de argumente, dar cu ACELAŞI tip de dată returnat. ATENŢIE: în acest caz noua funcţie NU VA MAI PUTEA FI MOŞTENITĂ MAI DEPARTE CA FIIND METODĂ VIRTUALĂ!

c. Evitarea mecanismului de apel uzual pentru funcţiile virtuale se face foarte uşor prin prefixarea numelui funcţiei cu numele clasei aparţinătoare urmat de "::".

Exemplul 3

```
void DERIVAT::f1(void)
{
 BAZA::tipareste_valoare(); // apelul metodei din clasa
                            //  BAZA
 tipareste_valoare(); // apelul metodei din clasa DERIVAT
}
```

d. Cu toate că nici un constructor nu poate fi declarat virtual, destructorii acceptă o astfel de prefixare. Ea poate fi deosebit de utilă în cazul în care se doreşte distrugerea uniformă a unor masive de date eterogene. O situaţie clară de utilizare a unor destructori virtuali vom obţine transformînd Exemplul 2 în următoarea manieră:

Exemplul 4

```
# include <iostream.h>

class BAZA {
 protected :
    int valoare;
 public :
    virtual void set_val(int a) { valoare=a; }

    virtual void tipareste_valoare(void)
    {
       cout<<"Element BAZA cu VALOARE = "
           <<valoare<<"\n";
    }
```



```
    virtual ~BAZA(void)
    {
       cout<<"\t\t DESTRUCTOR BAZA\n\n";
    }
};

class DERIVAT : BAZA {
 public :
    BAZA::set_val;

    void tipareste_valoare(void)
    {
       cout<<"Element DERIVAT cu VALOARE = "
           <<valoare<<"\n";
    }

    ~DERIVAT(void)
    {
       cout<<"\t\t DESTRUCTOR DERIVAT\n";
    }
};

class LISTA_ETEROGENA {
    BAZA * pB;
 public:

    BAZA* get(void)
    {
       return pB;
    }

    void set(BAZA * p)
    {
       pB=p;
    }

    void tipareste_valoare(void)
    {
       pB->tipareste_valoare();
    }
};
```

```
void main(void)
{
 LISTA_ETEROGENA  p[4];

 p[0].set(new BAZA);
 p[1].set((BAZA *) new DERIVAT);
 p[2].set((BAZA *) new DERIVAT);
 p[3].set(new BAZA);

 for(int i=0;i<4;i++)
 {
    (p[i].get())->set_val(i);
    p[i].tipareste_valoare();
    delete p[i].get();
 }
}
```

În urma execuţiei acestui program se vor afişa pe ecran următoarele date:

Element BAZA cu VALOARE = 0
DESTRUCTOR BAZA

Element DERIVAT cu VALOARE = 1
DESTRUCTOR DERIVAT
DESTRUCTOR BAZA

Element DERIVAT cu VALOARE = 2
DESTRUCTOR DERIVAT
DESTRUCTOR BAZA

Element BAZA cu VALOARE = 3
DESTRUCTOR BAZA

În acest mod avem confirmarea clară a corectitudinii distrugerii uniforme a listei eterogene de obiecte.

c. O funcţie virtuală "F", redefinită într-o clasă derivată "D", va fi vazută în noua sa formă de către toate clasele derivate din "D". Această redefinire nu afectează cu nimic faptul că ea este virtuală, metoda rămînînd în continuare virtuală şi pentru eventualele clase derivate din "D".

Exemplul 5

```
# include <iostream.h>

class B1 {
 public:
    virtual void t1(void) { cout<<"B1\n"; };
```



```
};

class B2 : public B1 {
 public :
    void t1(void) { cout<<"B2\n"; }
};

class B3 : public B2 {
};

void main(void)
{
 B1 a;
 B2 b;
 B3 c;

 a.t1();
 b.t1();
 c.t1();
}
```

Efectul execuţiei acestui exemplul va fi tipărirea pe ecran a următoarelor şiruri de caractere:

B1
B2
B2

Aşadar, în clasa "B3", funcţia virtuală "t1" este "vazută" aşa cum a fost ea redefinită în clasa "B2". Vechea formă (cea corespunzătoare clasei "B1") poate fi utilizată aşa cum am arătat la punctul "c." al observaţiilor.

f. Să considerăm exemplul de mai jos:

Exemplul 6

```
# include am.h

class BAZA {
 protected :
    virtual void tipareste_mesaj(void)
    {
       cout<<"\tMetoda apartine clasei BAZA\n";
    }

 public :
    void functie(void)
    {
```

```
        cout<<"\n\n URMEAZA UN MESAJ :\n";
            tipareste_mesaj();
    }
};

class DERIVAT : public BAZA {

 void tipareste_mesaj(void)
 {
    cout<<"\tMetoda apartine clasei DERIVAT\n";
 }
};

void main(void)
{
 BAZA    a;
 DERIVAT b;

 BAZA *  pB;

 a.functie();
 b.functie();

 pB=&a;
 pB->functie();

 pB=&b;
 pB->functie();
}
```

În urma execuţiei sale se va afişa următorul text:

URMEAZA UN MESAJ :
Metoda apartine clasei BAZA

URMEAZA UN MESAJ :
Metoda apartine clasei DERIVAT

URMEAZA UN MESAJ :
Metoda apartine clasei BAZA

URMEAZA UN MESAJ :
Metoda apartine clasei DERIVAT

Ce se observă în acest exemplu? Metoda"BAZA::functie" va fi moştenită

de către clasa DERIVAT. Mecanismul de "LATE BINDING" permite selectarea corectă a lui "DERIVAT::tipareste_valoare" la fiecare apel al lui "functie" pentru cîte o instanţiere a clasei DERIVAT.

g. Cînd se pune problema de a decide dacă o metodă să fie sau nu virtuală, programatorul va lua în considerare următoarele aspecte:
   - În cazul în care contează doar performanţele unui program şi se exclude intenţia de a-l mai dezvolta în continuare, SE VA OPTA PENTRU METODE NE-VIRTUALE.
   - În cazul programelor ce urmează să suporte dezvoltări ulterioare, situaţia este cu totul alta. Vor fi declarate virtuale toate acele metode ale clasei "CLASA" care se consideră că ar putea fi redefinite în cadrul unor clase derivate din "CLASA" şi, în plus, modificarea va trebui să fie sesizabilă la "nivelul" lui "CLASA"! Toate celelalte metode pot rămîne ne-virtuale.

4.6. Moştenire multiplă

Una din marile noutăţi ale versiunii AT&T 2.0 a limbajului C++ constă în posibilitatea ca o clasă să fie derivată din mai multe clase de bază. În versiunile precedente, relaţia de moştenire era limitată la doua clase: cea de BAZĂ şi cea DERIVATĂ. Cu toate acestea, moştenirea multiplă se putea simula destul de uşor:

Exemplul 1

```
class B1
{
 // ...
};

class B2
{
 // ...
};

class B3
{
 // ...
};

class D1 : B1
{
 B2 b1;
```

```
 B3 b2;
};
```

Din punct de vedere al structurii de date, clasa"D1" din exemplul de mai sus este echivalentă cu forma:

```
class D1 : B1 , B2 , B3
{
 // ...
};
```

Din punct de vedere funcţional, există totuşi destule diferenţe (modul de apel al metodelor claselor membre, dificultăţi în lucrul cu masive de date eterogene etc.). Tocmai de aceea, noua versiune elimină aceste neajunsuri, acceptînd existenţa relaţiei de moştenire multiplă. În acest mod, o clasă derivată va putea avea SIMULTAN mai multe clase de bază!

Există, totuşi, şi unele restricţii:

a. O clasă derivată "D" nu poate moşteni direct de două ori aceeaşi clasă "B". Motivul este foarte simplu: nu s-ar putea face distincţia între membrii purtînd acelaşi nume!

Exemplul 2

```
class B
{
 // ...
};

class D : B , B    // EROARE !!!!
{
};
```

b. O clasă derivată "D" nu poate moşteni simultan o clasă "B" şi o alta "D1", derivată (direct sau nu) tot din "B". De asemenea, nu pot fi moştenite simultan două clase "D1" şi "D2" aflate în relaţia de moştenire "B"->... ->"D1"->...->"D2".

Exemplul 3

```
class B
{
 // ...
};

class C1 : B
{
};
```



```
class C2 : C1
{
};

class D1 : B , C2    // EROARE
{
};

class D2 : C1 , C2   // EROARE
{
};
```

În schimb, o moştenire de genul

```
class B
{
 // ...
};

class D1 : B
{
};

class D2 : B
{
};


class D : D1 , D2
{
};
```

este găsită corectă de către compilatorul C++.

Pentru a realiza o moştenire multiplă, este suficient să specificăm după numele clasei derivate o listă de clase de bază separate prin virgulă (evident, aceste nume pot fi prefixate cu modificatorii de acces "public" sau "private"). Dar atenţie: ordinea claselor de bază nu este indiferentă! Apelul constructorilor respectivi se va face EXACT în ordinea enumerării numelor claselor de bază! Reluînd observaţia de la moştenirea simplă, apelul constructorilor tuturor claselor de bază se va efectua ÎNAINTEA eventualelor iniţializări de variabile-membru. Mai mult, aşa cum vom vedea în exemplul de mai jos, această ordine nu poate fi modificată nici chiar de ordinea apelului explicit al constructorilor claselor de bază în cadrul constructorului clasei derivate.

Exemplul 4

```
class B1
{
    char c;

 public :
    B1(char c1) : c(c1) {};
};

class B2
{
    int c;

 public :
    B2(int c1) : c(c1) {};
};

class D : B1 , B2
{
    char a,b;

 public :
    D(char a1,char b1) : a(a1) , b(b1)
      B2((int) a1) , B1(b1)
 {
    // ...
 }
};

void main(void)
{
 D d(3,3);
}
```

După cum am mai afirmat, indiferent de ordinea iniţializării variabilelor şi de cea a constructorilor specificaţi explicit, în definirea constructorului clasei derivate

```
D(char a1,char b1) : a(a1) , b(b1)
      B2((int) a1) , B1(b1) {};
```

succesiunea operaţiilor va fi următoarea: apelul constructorului "B1", apoi "B2" şi, doar la sfîrşit, iniţializarea lui "a" şi "b".

O situaţie cu totul particulară o au aşa-zisele CLASE VIRTUALE.

Pentru a avea o imagine cît mai clară asupra problemei, să analizăm exemplul de mai jos:

Exemplul 5

```
class B
{
};

class D1 : B
{
};

class D2 : B
{
};

class M1 : D1 , public D2
{
};

class M2 : virtual D1 , virtual public D2
{
};
```

În urma moştenirii claselor "D1" şi " D2", clasa "M1" va îngloba două obiecte de tipul "B" (cîte unul pentru fiecare din cele două clase de bază). În practică, ar putea exista situaţii în care este de dorit moştenirea unui singur obiect de tip "B". Pentru aceasta se va proceda ca şi în cazul clasei "M2": se vor prefixa cele două clase de bază cu cuvîntul cheie "virtual". În urma acestei decizii, clasa "M2" nu va conţine decît O SINGURĂ INSTANŢIERE A CLASEI "B".

Deşi în aparenţă totul e deosebit de simplu, se mai poate pune o întrebare: cărei clase îi va aparţine acea unică instanţiere? Lui D1 sau lui D2? INSTANŢIEREA ÎN CAUZĂ VA APARŢINE LUI "D1", adică PRIMEI CLASE VIRTUALE (din lista claselor de bază) care a fost derivată din "B". În urma analizei exemplului de mai jos veţi remarca un fapt destul de ciudat, pe care-l vom comenta în continuare.

Exemplul 6

```
class B
{
 public:
     int i;
};
```

```
class D1 : public B
{
};

class D2 : public B
{
};

class M1 : public D1 , public D2
{
};

class M2 : virtual public D1 , virtual public D2
{
};

class M3 : virtual public D2 , virtual public D1
{
};

void main(void)
{
 M1 m1;
 M2 m2;
 M3 m3;

 m1.D1::i=0;
 m1.D2::i=1;

 m1.i=0; // EROARE: exprimare ambigua ( de care din cei
         // doi membri "i" e vorba ? )

 m2.D1::i=4;      // CORECT

 m2.D2::i=5;      // ATENTIE: desi nu se semnalizeaza eroare la
                  // compilare, aceasta instructiune NU
                  // ARE NICI UN EFECT !!!

 m2.i=6;          // CORECT: e echivalent cu "m2.D1::i= 6;"

 m3.D1::i=7;      // ATENTIE: desi nu se semnalizeaza eroare la
                  // compilare, aceasta instructiune NU
                  // ARE NICI UN EFECT !!!
```



```
 m3.D2::i=8;      // CORECT

 m3.i=9;          // CORECT: e echivalent cu "m3.D2::i= 9;"
}
```

Concluzia este următoarea: deşi conform regulii enunţate mai sus variabila membru aparţine instanţierii primei clase din lista claselor de bază, cele două forme "m2.D2::i" şi "m3.D1::i" nu vor fi tratate de către compilator ca şi erori. În schimb, efectul lor nu va fi cel scontat, adică atribuirile în cauză nu vor fi efectuate (instrucţiunile respective neavînd deci nici un efect). Atenţie deci la această sursă de erori! Cea mai simplă modalitate de a evita astfel de neplăceri constă în a renunţa la prefixarea variabilei membru cu clasa de care aparţine. În acest mod nu pot apărea ambiguităţi, deoarece variabila este unică.

Definirea claselor virtuale ne obligă să modificăm regulile de apelare a constructorilor claselor de bază. În cele ce urmează vom prezenta noul set de reguli:

R1. *Constructorii claselor de bază virtuale vor fi apelaţi ÎNAINTEA celor corespunzînd unor clase de bază ne-virtuale.*

R2. *În caz că o clasă posedă mai multe clase de bază virtuale, constructorii lor vor fi apelaţi în ordinea în care clasele au fost declarate în lista claselor de bază. Apoi vor fi apelaţi (tot în ordinea declarării) constructorii claselor de bază ne-virtuale, şi abia în cele din urmă constructorul clasei derivate.*

R3. *Chiar dacă în ierarhia claselor de bază există mai mult de o instanţiere a unei clase de bază virtuale, constructorul respectiv va fi apelat doar o singură dată.*

R4. *Dacă, în schimb, în aceeaşi ierarhie există atît instanţieri virtuale cît şi ne-virtuale ale unei aceleiaşi clase de bază, constructorul va fi apelat:*

O SINGURĂ DATĂ	– pentru toate instanţierile virtuale;
DE "N" ORI	– pentru cele ne-virtuale (dacă există "N" instanţieri de acest tip).

Pentru a clarifica lucrurile vom oferi un exemplu mai complex, pe care vom studia modul de aplicare a acestor reguli.

Exemplul 7

```
int i=0;

class B1
{
 public:
    B1(void) {
       i++;
```

```
    }
};

class B2
{
 public:
    B2(void) {
        i++;
    }
};

class D1 : public B1 , virtual public B2
{
 public:
    D1(void) {
        i++;
    }
};

class D2 : public B1 , virtual public B2
{
 public:
    D2(void) {
        i++;
    }
};

class M : public D1 , virtual public D2
{
 public:
    M(void) {
        i++;
    }
};

void main(void)
{
 M m1;
}
```

Ordinea în care vor fi apelaţi constructorii este următoarea:

```
B2();
B1();
D2();
B1();
D1();
M();
```

Ea corespunde întru totul regulilor enunţate mai sus. În încercarea de a construi o instanţiere a clasei "M", se vor analiza cele două clase de bază. Deoarece "D2" este clasa de bază virtuală, se va începe prin a crea un obiect de tip "D2" (regula "R1"). Pentru aceasta va fi necesară crearea unor obiecte de tip "B1" şi "B2". Se va începe cu "B2" deoarece este vorba tot de o clasă de bază virtuală (aceeaşi regulă "R1"). Din aceste motive, primul constructor apelat va fi "B2", urmat apoi de "B1". În acest moment va putea fi apelat şi constructorul "D2", deoarece au fost deja create instanţierile claselor sale de bază (regula "R2"). În fine, pentru a crea un obiect de tipul "D1" ar trebui să se înceapă cu crearea unui obiect de tip "B2". Cum clasa de bază "B2" este virtuală, şi, în plus, există deja o instanţiere a acestei clase, se va renunţa la crearea obiectului de tip "B2" (regula "R3"). Se va continua, deci, cu apelul constructorului clasei de bază "B1", urmat apoi de constructorul "D1" (regula "R2"). Deoarece în acest moment sînt create instanţierile claselor de bază "D1" şi "D2", se va putea apela şi constructorul clasei "M" (tot regula "R2").

În încheierea secţiunii prezentăm încă un exemplu de funcţionare a mecanismului de moştenire multiplă.

Exemplul 8

```
# include <graphics.h>
# include <conio.h>

class arc_de_cerc
{
    int xc,yc,raza,unghi_start,unghi_sfirsit;
 public:
    arc_de_cerc(int x,int y,int r,int u1, int u2)
    {
        xc=x;
        yc=y;
        raza=r;
        unghi_start=u1;
        unghi_sfirsit=u2;

        arc(xc,yc,unghi_start,unghi_sfirsit,raza);
    }
}
```

```
class linie
{
    int x1,x2,y1,y2;
 public:
    linie(int a,int b,int c,int d)
    {
        x1=a;
        y1=b;
        x2=c;
        y2=d;

        line(x1,y1,x2,y2);
    }
};

class cap : public arc_de_cerc
{
 public :
    cap(int xc,int yc,int u_st,int u_sf,int raza)
     : arc_de_cerc(xc,yc,u_st,u_sf,raza)
    {
    }
};

class ochi_stinga  : public arc_de_cerc
{
 public :
    ochi_stinga(int xc,int yc,int u_st,int u_sf,
     int raza) : arc_de_cerc(xc,yc,u_st,u_sf,raza)
    {
    }
};

class ochi_dreapta : public arc_de_cerc
{
 public :
    ochi_dreapta(int xc,int yc,int u_st,int u_sf,
     int raza) : arc_de_cerc(xc,yc,u_st,u_sf,raza)
    {
    }
};
```



```
class gura : public arc_de_cerc
{
 public :
    gura(int xc,int yc,int u_st,int u_sf,
     int raza) : arc_de_cerc(xc,yc,u_st,u_sf,raza)
    {
    }
};

class fata_umana : cap,ochi_stinga,
                   ochi_dreapta,gura,linie
{
 public:
    fata_umana(void) : cap(100,100,50,0,360),
                linie(100,100,100,110),
                ochi_stinga(80,90,3,0,360),
                ochi_dreapta(120,90,3,0,360),
                gura(100,100,40,240,300)
    {}
};

void main(void)
{
 int gd,gm;

 detectgraph(&gd,&gm);
 initgraph(&gd,&gm,"\\compiler\\t_pascal\\bgi");

 fata_umana f;

 getch();
 closegraph();
}
```

Neputînd forţa clasa "fata_umana" să moştenească de patru ori clasa "arc_de_cerc", vom recurge la un subterfugiu: din ea vom deriva alte patru clase ("cap", "ochi_stinga", "ochi_dreapta" şi "gura"), pe care le vom utiliza ca şi clase de bază ale clasei "fata_umana".

4.7. Problemă rezolvată. Exerciţii propuse spre rezolvare

În cele ce urmează, vom prezenta un exemplu care are drept scop fa-

miliarizarea cu situaţii de o complexitate mai apropiată de cea reală. Ideea programului este de a implementa o familie de ferestre grafice, care să faciliteze şi să facă mai plăcut dialogul cu utilizatorul.

În esenţă, programul constă într-un PROJECT compus din 4 module: clasele "settings", "general_window", "error_window" şi fişierul "main.cpp" (conţine programul principal care utilizează cele trei clase). Vom începe să analizăm pe rînd fişierele în cauză, primul fiind "header"-ul clasei "settings":

/*FISIERUL STTNG_CL.DAT */

```
class settings {

 int cul,bkcul;       // culoare de scriere si de fond
 struct textsettingstype txt; // atributele de scriere ale
                              // textelor
 struct fillsettingstype fll; // atributele de umplere

protected:
     void put_settings(void); // setare atribute mod grafic
     void get_settings(void); // salvare atribute mod grafic

};
```

Acest "header" va fi inclus în fişierul "SETTINGS.CPP", care conţine definiţiile metodelor declarate în "STTNG_CL.DAT". Clasa "settings" are ca scop implementarea unor rutine de salvare/restaurare a caracteristicilor modului de lucru grafic (culoare de desenare şi fond, caracteristici de scriere şi de umplere).

/*FISIERUL SETTINGS.CPP */

```
# include <graphics.h>
# include "sttng_cl.dat"

// Restaureaza atributele modului de lucru salvate anterior
void settings::put_settings(void)
{
 settextstyle(txt.font,txt.direction,txt.charsize);
 settextjustify(txt.horiz,txt.vert);

 setfillstyle(fll.pattern,fll.color);
 setcolor(cul);
 setbkcolor(bkcul);
}
```



```
// Salveaza atributele curente de lucru in mod grafic
void settings::get_settings(void)
{
 gettextsettings(&txt);
 getfillsettings(&fll);
 cul=getcolor();
 bkcul=getbkcolor();
}
```

În continuare, vom analiza descrierea unei clase numită "GENERAL_WINDOW", din care să putem deriva mai apoi toate tipurile de ferestre dorite. Vom începe cu "header"-ul care conţine declaraţia clasei:

/*FISIERUL G_WIN_CL.DAT */

```
# include "sttng_cl.dat"

# define MARGINE 8

class general_window : settings {

private:
//.....VARIABILE MEMBRU.....

 int x,y,dx,dy; // coordonatele colturilor stinga-sus si
                // dreapta-jos ale ferestrei

 void * store_image; // pointer la imaginea peste care
                     // va fi desenata fereastra

 int image_size;    // lungimea (octeti) zonei de mai sus

 char * window_name; // pastreaza numele ferestrei

 int name_size; // lungimea numelui ferestrei (nr. caractere)

//.....METODE.....

 void set_xy(void); // initializeaza coordonatele colturilor
                    // ferestrei
 void get_image(void); // salvare imagine de sub fereastra
 void draw_window(void); // desenare fereastra
```

```
    int bad_coord(int,int,int,int); // verifica coord. colturi

  protected :

    void my_settings(int,int); // seteaza caracteristici mod
                               // de lucru grafic

    settings::get_settings;    // forteaza la "protected" cele
    settings::put_settings;    // doua metode mostenite

  public :

    void create_window(char *,int=-1,int=-1,
                       int=-1,int=-1);
    void destroy_window(void);
  };
```

Fişerul "GNRL_WIN.CPP" va conţine definiţiile metodelor clasei "general_window".

/*GNRL_WIN.CPP*/

```
  # include <iostream.h>
  # include <graphics.h>
  # include <process.h>
  # include <string.h>

  # include "g_win_cl.dat" // contine declaratia clasei

  // Salvare imagine de "sub" fereastra
  void general_window::get_image()
  {
    image_size=imagesize(x,y,dx,dy);
    store_image=new char[image_size];
    if (store_image==NULL) {
       cout<<"Out of memory in GENERAL_WINDOW\n";
       exit(1);
    }
    getimage(x,y,dx,dy,store_image);
  }

  // Setarea caracteristicilor modului de lucru grafic
  void general_window::my_settings(int bk_color,
                                   int color )
```



```
{
// APEL METODA MOSTENITA "get_settings()" !!!
 get_settings(); // salvarea caracteristici mod grafic

 settextstyle(DEFAULT_FONT,HORIZ_DIR,1);
 settextjustify(LEFT_TEXT,LEFT_TEXT);

 setfillstyle(SOLID_FILL,bk_color);
 setcolor(color);
}

// Desenare fereastra
void general_window::draw_window()
{
// afisarea numelui ferestrei
 bar(x,y,dx,y+9);
 outtextxy((dx-x)/2+x-textwidth(window_name)/2,
           y+9,window_name);

// desenarea corpului ferestrei
 setcolor(WHITE);
 setfillstyle(SOLID_FILL,BLACK);
 bar(x,y+9,dx,dy);
 rectangle(x,y+10,dx,dy);
 rectangle(x+2,y+12,dx-2,dy-2);
}

// Seteaza abscisele si ordonatele colturilor stinga-sus
// (x,y) si dreapta-jos (dx,dy) la valorile implicite
void general_window::set_xy(void)
{
 x=getmaxx()/2-150;
 y=getmaxy()/2-80;
 dx=getmaxx()/2+150;
 dy=getmaxy()/2+80;
}

// Verifica daca ambele colturi sint in cadrul ecranului si
// daca intr-adevar coltul stinga-sus este mai sus si mai in
// stinga decit cel dreapta-jos
int general_window::bad_coord(int x1,int y1,
                              int x2,int y2)

{
```

```
  if( x1<0 || y1<0 || x2<0 || y2<0 || x2>getmaxx()
     || y2>getmaxy() || x1>=x2 || y1>=y2 )
  return 1;

  return 0;
}

// Va crea o fereastra cu numele dat de "window_name" si cu
// coordonatele colturilor stinga-sus si dreapta-jos luind
// valorile parametrilor "i1"-"i4" (eventual se pot utiliza
// valorileimplicite);
void general_window::create_window(char *win_name,
                                   int x1, int y1,
                                   int x2,int y2)
{
 if( bad_coord(x1,y1,x2,y2) )
// setare coord. colturi la val. implicite
     set_xy();
 else {
// init. coordonatelor la valorile transmise ca parametri
     x=x1;
     y=y1;
     dx=x2;
     dy=y2;
 }

 get_image();      // salvare imagine de "sub" fereastra

 my_settings(RED,BLACK);   // seteaza caracteristicile de
                           // lucru in mod grafic

 int lung=textwidth(win_name);

// daca numele fereastrei nu incape in fereastra, fereastrei
// nu i se va afisa numele
 if (dx-x-2*MARGINE>=lung) {
// numele incape in fereastra
     name_size=strlen(win_name)+1;
     window_name=new char[name_size];
     strcpy(window_name,win_name);
 }
 else {
```



```
// numele nu incape in fereastra
     name_size=2;
     window_name=new char[name_size];
     strcpy(window_name," ");
 }

 draw_window();   // desenare fereastra

 put_settings(); // restaurare optiuni utilizator
}

// - restaurare caracteristici mod de lucru grafic folosite de utilizator;
//  - refacere imagine de "sub" fereastra;
// - eliberare memorie consumata pentru pastrarea imaginii salvate.
void general_window::destroy_window(void)
{
 put_settings();
 putimage(x,y,store_image,COPY_PUT);
 delete[image_size] store_image;
 delete[name_size] window_name;
}
```

În fine, clasa "ERROR_WINDOW" va avea următorul "header":

/* E_WIN_CL.DAT */

```
# include "g_win_cl.dat"

class error_window : general_window {
 public :
     void create_error_window(char *);
     void destroy_error_window(void);
};
```

Definirea metodelor respective se va face în fişierul "ERR_WIN.CPP":

/* ERR_WIN.CPP */

```
# include <graphics.h>
# include <process.h>
# include <conio.h>

# include "e_win_cl.dat" // contine declaratia clasei

# define MARGINE 8
```

```
void error_window::create_error_window(char *
                        error_msg)
{
 char * mesaj,msg[]="Mesaj de eroare prea lung !";
 int lung,inalt,x=getmaxx()/2,y=getmaxy()/2;

// setare caracteristici de lucru
 my_settings(BLACK,WHITE);

 lung=textwidth(error_msg)/2;
 inalt=textheight(error_msg);
 mesaj=error_msg;

 // daca mesajul este prea lung va fi inlocuit cu unul de avertizare
 if(lung+MARGINE>x) {
     lung=textwidth(msg);
     inalt=textheight(msg);
     mesaj=msg;
 }

// inainte de a crea fereastra se refac vechile caracteristici
// (pentru a nu fi pierdute prin supra-inscriere)
 put_settings();

 create_window("EROARE",x-lung-MARGINE,y-2*inalt,
               x+lung+MARGINE,y+3*inalt);

 my_settings(BLACK,WHITE);
 outtextxy(x-lung,y+2*inalt,mesaj);
 put_settings();
}

void error_window::destroy_error_window()
{
 destroy_window();
}
```

Referitor la metoda "error_window::destroy_error_window()", este clar că ea putea lipsi, în locul ei utilizindu-se "destroy_window()"-ul moştenit. S-a preferat această soluţie deoarece ni se pare mai elegant ca numele metodei să fie cît mai sugestiv.

În fine, fişierul "MAIN.CPP" arată astfel:

/* MAIN.CPP */

```
# include <conio.h>
# include <stdio.h>
# include <process.h>
# include <graphics.h>

# include "e_win_cl.dat"

// Asigura initializarea modului de lucru grafic si furnizarea
// mesajelor de avertizare si eroare corespunzatoare
void init_graph(void)
{
 int gd,gm,err;

 detectgraph(&gd,&gm);
 initgraph(&gd,&gm,"\\borlandc\\bgi");
 err=graphresult();
 if(err!=grOk) {
    printf("GRAPHICS ERROR : %s\n",
          grapherrormsg(err));
    exit(1);
 }
}

// Pe ecran se va crea un desen oarecare (pentru a se arata ca
// utilizarea ferestrelor nu distruge imaginea existenta)
void creaza_scena()
{
 char txt[] ="Este";
 char txt1[]="un program";
 char txt2[]="demonstrativ al cartii de C++";

 circle(50,50,50);
 circle(80,80,80);
 circle(120,120,120);

 settextjustify(LEFT_TEXT,LEFT_TEXT);

 settextstyle(TRIPLEX_FONT,HORIZ_DIR,10);
 outtextxy((getmaxx()/2)-(textwidth(txt)/2),
           (getmaxy()/3)+(textheight(txt)/2),txt);
```

```
 settextstyle(TRIPLEX_FONT,HORIZ_DIR,2);
 outtextxy((getmaxx()/2)-(textwidth(txt1)/2),
           getmaxy()-2*textheight(txt1),txt1);

 settextstyle(GOTHIC_FONT,HORIZ_DIR,4);
 outtextxy((getmaxx()/2)-(textwidth(txt2)/2),
           getmaxy()-2,txt2);
}

// Aceasta procedura va crea si manipula diverse tipuri de ferestre,
// incercind sa surprinda situatii cit mai complexe (suprapuneri partiale
// sau totale, etc.)
void actiune(void)
{
 general_window gw1;
 general_window gw2;
 general_window gw3;

 gw1.create_window("Fereastra numarul 1");
 outtextxy(500,30,"A");
 gw2.create_window("Numarul 2",10,10,100,100);
 outtextxy(500,60,"B");
 gw3.create_window("Numarul 3",50,50,150,150);
 outtextxy(500,90,"C");

 getch();

 gw3.destroy_window();
 getch();
 gw2.destroy_window();
 getch();
 gw1.destroy_window();
 getch();

 outtextxy(500,120,"D");

 gw1.create_window("Fereastra numarul 1");
 gw2.create_window("Numarul 2",10,10,100,100);
 gw3.create_window("Numarul 3",50,50,150,150);

 error_window ew1;
 ew1.create_error_window("Disc protejat la
                          scriere !");
```



```
 outtextxy(500,150,"E");
 getch();

 error_window ew2;
 ew2.create_error_window("Memorie insuficienta !");
 outtextxy(500,180,"F");
 getch();

 ew2.destroy_error_window();
 getch();
 ew1.destroy_error_window();
 getch();
}

void main(void)
{
 init_graph();

 creaza_scena();
 actiune();

 closegraph();
}
```

Ca probleme propuse spre rezolvare, am sugera extinderea acestui set de ferestre, după cum urmează:

a. crearea unui mecanism de redimensionare şi schimbare a poziţiei ferestrei generale pe ecran;
b. crearea unei ferestre de prompt (citire de caractere, numere întregi şi reale);
c. crearea unei ferestre de căutare pe disc a unor clase de fişiere (Ex: *.*, *.txt, a?n.*, ...).

5. ALTE FACILITĂŢI ALE LIMBAJULUI C+ +

5.1. Funcţii pure şi clase abstracte

Începem cu această temă deoarece este direct legată de unul dintre subiectele precedentului capitol, şi anume de mecanismul de moştenire a metodelor virtuale. Vom reaminti în cîteva cuvinte despre ce era vorba. Dacă o metodă "M" a unei clase de baza "B" este prefixată cu cuvîntul cheie "virtual", ca va fi utilizată identic de toate clasele "MB" ce o moştenesc pe "B" fără a redefini metoda "M". Dacă o clasă "MB1", derivată din "B", va redefini metoda "M", compilatorul va şti să utilizeze noua metodă indiferent dacă ea va fi manipulată:

a. de un obiect de tipul MB1;
b. prin intermediul unui pointer la obiecte de tipul B, care memorează adresa unui obiect de tip MB1.

Oricum, este clar că din moment ce pot fi apelate, metodele virtuale vor fi atît declarate, cît şi definite.

În unele situaţii practice se poate găsi o clasă "B" din care se derivează toate celelalte clase ale ierarhiei. Uneori clasa "B" este atît de generală, încît unele metode ce reprezintă caracteristici fundamentale ale ei nu pot fi descrise la acest nivel, ci doar în clasele derivate.

De exemplu, fie următoarea ierarhie de clase:

VIEŢUITOARE:

- LALELE
- NUFERI
- CAI
- OAMENI

Să presupunem că în cadrul clasei VIEŢUITOARE există o metodă numită NUTRIŢIE. La acest nivel, funcţia NUTRIŢIE nu poate fi definită decît abstract. Deşi putem spune că NUTRIŢIA este "totalitatea proceselor de ingerare şi asimilare a hranei" ([MDLR], p. 488), această definiţie nu ne dă nici o informaţie despre modalităţile practice de realizare a procesului respectiv. NUTRIŢIA va fi însă foarte uşor de definit concret în cadrul claselor derivate. Spre exemplu, procesele de tipul masticaţiei sau digestiei

omeneşti pot fi descrise pînă în cele mai mici amănunte. Cînd spunem "O VIEŢUITOARE SE HRĂNEŞTE", ne vom închipui un nufăr, un cal sau un om. Deci, cînd spunem "O VIEŢUITOARE", ne referim la o entitate abstractă (deoarece oamenii, caii şi nuferii nu sînt instanţieri ale clasei VIEŢUITOARE). Altfel spus, nu există instanţieri ale clasei VIEŢUITOARE, ci doar obiecte aparţinînd unor clase derivate din aceasta.

Revenind la limbajul C+ +, vom spune că, prin definiţie, funcţiile de tipul NUTRIŢIE se vor numi FUNCŢII PURE. Deci, o FUNCŢIE PURĂ este o METODĂ VIRTUALĂ ce nu poate fi DEFINITĂ (dar va fi obligatoriu DECLARATĂ) la un anumit nivel de abstractizare.

Odată definite FUNCŢIILE PURE, vom spune că o CLASĂ ABSTRACTĂ este o clasă care are cel puţin o metodă pură. O clasă de tipul VIEŢUITOARE va fi o CLASĂ ABSTRACTĂ tocmai fiindcă NU POT EXISTA INSTANŢIERI ALE EI. Acest lucru se datorează imposibilităţii de a defini funcţia de NUTRIŢIE.

Clasele abstracte se pot utiliza absolut identic cu cele ne-abstracte, cu unica diferenţă că NU POT EXISTA INSTANŢIERI ALE LOR.

Urmărind exemplul de mai jos, totul se va clarifica pînă în cele mai mici amănunte.

Exemplul 1

```
# include <process.h>
# include <dos.h>   // contine prototipul functiilor "sound",
                    //  "nosound" si "delay"

class joc{
 public :

// POST FIXAREA unei declaratii de metoda cu "=0" va determina
//  ca respectiva metoda sa devina PURA

 virtual void demo(void) = 0 ;     // FUNCTIE PURA
 virtual void melodie(void) = 0 ; // FUNCTIE PURA
};

class samurai : public joc{
 public :
    void demo(void)
    {
        system("samurai /d");
    }
```

```
    void melodie(void);
};

void samurai::melodie(void)
{
 for(int i=0;i<10;i++)
 {
    sound(1000+100*i); // provoaca emiterea unui sunet de
                       // frecventa egala cu valoarea para-
                       //metrului actual
    delay(500);    // opreste executia programului timp de 500
                   // milisecunde (in acest mod se stabileste
                   // durata sunetului emis)
    nosound();         // opreste emiterea sunetului
 }
}

void main(void)
{
 samurai s;

 s.melodie();
}
```

De notat că dacă în declararea clasei "samurai" am fi avut secvenţa

```
class samurai : public joc
{
 public :
    void demo(void) = 0;
    //...
}
```

atunci "samurai" ar fi rămas o clasă abstractă, avînd însă o singură metodă pură ("demo"). Atenţie deci: în cadrul ierarhiei de clase, orice funcţie pură trebuie ori să fie redeclarată pură, ori să fie definită!!!

5.2. OVERLOADING OPERATORS

Termenul englezesc de "OVERLOADING OPERATORS" se referă la o facilitate specifică limbajului C++: dreptul programatorului de a adăuga operatorilor limbajului şi alte sensuri decît cele predefinite. Cu alte cuvinte, este vorba de o redefinire a operatorilor C++, fără ca vechile sensuri să se piardă.

Se ştie că operatorul "+" poate fi utilizat pentru a realiza adunări de numere întregi şi reale. Prin "OVERLOADING" (o traducere destul de plastică ar fi SUPRAÎNCĂRCARE), operatorul "+" ar mai putea fi folosit la adunarea de matrici, numere complexe sau raţionale. Cum s-ar putea realiza acest lucru? Foarte simplu: claselor MATRICE, COMPLEX sau RAŢIONAL li se va adăuga o metodă-operator numită "+". Evident, întreaga grijă pentru o utilizare corectă a operatorului cade în grija compilatorului. Astfel, presupunînd că am definit clasele COMPLEX şi RAŢIONAL că avînd un operator "+", în secvenţa de mai jos totul va decurge după cum dorim:

```
int        a=0,b=6;
double     c=4.5,d=6.7;
COMPLEX    e(2,3),f(3,4);
RATIONAL   g(4,5),h(5,6);

// .........

a=a+b; // adunare de intregi
c=c+d; // adunare de reali
e=e+f; // adunare de numere complexe
g=g+h; // adunare de numere rationale
```

Din punct de vedere al limbajului, operatorii redefiniţi nu sînt altceva decît nişte metode oarecare, purtînd însă nume de operatori (de exemplu metodele cu numele +, -, etc). Pentru a nu provoca erori de sintaxă, numele metodelor în cauză vor fi prefixate cu cuvîntul cheie "operator". Să considerăm următoarea secvenţă:

```
class matrice
{
// ...
 public :
    matrice& AdunaMatrici(matrice&);
    matrice& operator + (matrice&);
// ...
};
```

Unica diferenţă între cele două metode constă în comoditatea sporită a apelului celei de a doua:

```
matrice a,b,c;
// ...
c=a.AdunaMatrice(b); // INCOMOD
c=a.operator+(b); // CORECT, DAR LA FEL DE INCOMOD
c=a+b;            // MULT MAI COMOD SI MAI NATURAL
```

Înainte de a încheia această secţiune, vom mai face cîteva observaţii deosebit de importante. Deşi aproape oricărui operator i se poate da un nou sens, NICI UNUI OPERATOR NU I SE POATE SCHIMBA PRECEDENŢA SAU SINTAXA (un operator binar nu poate fi făcut unar şi nici invers)!!!!

De asemenea, nu este permis să se definească (introducă) noi operatori. E o restricţie absolut normală, lipsa ei putînd da naştere unor complicaţii atît de mari încît compilatorul să nu le mai poată rezolva.

Singurii operatori care nu pot fi redefiniţi sînt următorii:

. .* :: ?:

Reamintim că operatorul ".*" este utilizat la manipularea pointerilor la metode (vezi "3.4.").

ATENŢIE : directivele preprocesor # şi ## nu pot fi redefinite deoarece ELE NU SÎNT OPERATORI!

O altă chestiune importantă ţine de libertatea de redefinire a operatorilor. Pentru ca ea să fie totală, compilatorul NU VA FACE NICI O PRESUPUNERE LEGATĂ DE UN SENS PREDEFINIT AL OPERATORILOR. După cum sublinia şi Stroustrup ([STRO], p. 172), în încercarea de a oferi programatorului o maximă deschidere, compilatorul NU VA VERIFICA nici măcar în cazul redefinirii lui "=" DACĂ PRIMUL OPERAND ESTE SAU NU O "lvalue" (adică dacă i se pot atribui valori).

De asemenea, trebuie avut grijă la următorul aspect: în limbajul C expresiile

```
index++;
index+=1;
index=index+1;
```

erau echivalente. În C++, redefinirea lui "+" şi "=" nu e suficientă pentru ca acest lucru să fie adevărat. Trebuie ca programatorul să definească corespunzător TOŢI operatorii în cauză.

5.3. Operatori unari şi binari

Un lucru pe care nu l-am spus pînă acum este faptul că un operator poate fi nu doar metodă, ci şi funcţie "friend". Poate mai puţin aşteptată va fi IMPOSIBILITATEA de a defini un operator ca fiind METODĂ STATICĂ!!!

Dacă operatorul va fi definit ca metodă, există două posibilităţi:

este operator UNAR – în acest caz el nu va mai avea nici un argument (operatorul unar are nevoie de un singur operand: obiectul care va apela metoda);

este operator BINAR – situaţie în care metoda va avea un singur parametru (care reprezintă cel de al doilea operand).

Dacă operatorul este o funcţie "friend", vom avea alte două posibilităţi, după cum acesta va fi:

UNAR – situaţie în care va avea un singur parametru;

BINAR – şi atunci va avea doi parametrii.

Tot legat de operatorii unari şi binari: fie "@" şi "#" cîte un simbol reprezentînd un operator unar, respectiv binar. Atunci, atît expresiile "a@" şi "@a", cît şi expresia "x#y" pot avea cîte două interpretări, respectiv

```
a.operator@();
operator@(a);
```

şi

```
x.operator#(y);
operator#(x,y);
```

după cum operatorii "@" şi "#" vor fi metode sau funcţii "friend". Dacă în vreunul din cazuri au fost declarate ambele forme, compilatorul va decide apelul corect în funcţie de ordinea parametrilor în cele două funcţii operator.

Trebuie să facem următoarea subliniere: orice operator care e funcţie "friend" a unei clase "B" trebuie să aibă CEL PUŢIN un argument de tipul "B" (instanţiere a clasei "B"). În caz contrar, ar exista riscul de a introduce ambiguităţi în cadrul expresiilor formate doar din termeni de tipuri predefinite. Unica excepţie de la această regulă o constituie operatorii "new" şi "delete".

Cu titlul de remarcă, amintim faptul că formele pre şi post fixate ale operatorilor redefiniţi "++" şi "--" nu vor putea fi deosebite de către compilator.

O altă observaţie, oarecum neaşteptată, este cea legată de faptul că deşi operatorii moştenesc SINTAXA (unar/binar) şi PRECEDENŢA operatorilor predefiniţi, ei nu vor moşteni şi COMUTATIVITATEA lor. Exemplul de mai jos va sublinia foarte bine acest lucru:

Exemplul 1

```
class exemplu_1{
    int a;
 public :
    exemplu_1(int a1) : a(a1) {}
```

```
    friend int operator+(int i,exemplu_1 e)
    {
        return i+e.a;
    }

    friend int operator+(exemplu_1 e,int i)
    {
        return i+e.a;
    }
};

class exemplu_2{
    int a;
 public :
    exemplu_2(int a1) : a(a1) {}

    int operator+(int i)
    {
        return i+a;
    }
};

void main(void)
{
 exemplu_1 e1(10);
 exemplu_2 e2(100);
 int a;

 a=e1+1;     // CORECT
 a=1+e1;     // CORECT

 a=e2+1;     // CORECT
 a=1+e2;     // ERONAT: ordinea parametrilor nu se potriveste
}
```

Ce se remarcă în acest scurt program? În primul rînd frapează faptul că, în ciuda aşteptărilor, compilatorul nu va ţine cont de comutativitatea adunării, şi va semnaliza în consecinţă o eroare în linia

```
a=1+e2;
```

În aceeaşi ordine de idei, dacă am elimina unul dintre cei doi operatori ai clasei "exemplu_1", am mai obţine încă o eroare de compilare.

Cea de a doua observaţie va evidenţia un avantaj al operatorilor "friend"

faţă de cei de tip metodă: nu se poate defini un operator comutativ care să fie metodă şi să opereze cu operanzi de tipuri diferite (deoarece implicit primul operand va fii întotdeauna obiectul care apelează metoda). În schimb, acest lucru va fi uşor realizabil prin mecanismul de funcţii "friend".

Vom prezenta acum încă două exemple de redefinire a unor operatori. În cîteva cuvinte, despre ce este vorba: vom defini o clasă "rational", care va simula o parte dintre operaţiile cu numere raţionale. În exemplele de mai jos s-au implementat operaţiile de înmulţire (operaţie binară cu simbolul "*") şi inversa (operaţie unară cu simbolul "~": se va inversa numitorul şi numărătorul, semnalizînd eventualele împărţiri cu 0). ATENŢIE: deşi operaţia de înmulţire este corect efectuată, nu se operează eventualele reduceri posibile între numitor şi numărător. În Exemplul 2 operatorii vor fi metode, iar în Exemplul 3 funcţii "friend".

Exemplul 2

```
# include <iostream.h>
# include <conio.h>

class rational{
    int numitor,numarator;
 public:
    rational(int,int);

    void tip(char *);

    rational& operator*(rational&);
    rational& operator~(void);
};

rational::rational(int n1,int n2)
{
 numarator=n1;

// NU SE ACCEPTA NUMITORUL 0
 numitor = n2!=0 ? n2 : 1;
}

void rational::tip(char * nume)
{
 int a;
 cout<<" Nr. rational   "<<nume<<" = "<<numarator
     <<"/"<<numitor<<"\n";
}
```

```
rational& rational::operator*(rational& n)
{
 rational * r;
 int n1,n2;

 n1=numarator*n.numarator;
 n2=numitor*n.numitor;

 r=new rational (n1,n2);

 return *r;
}

rational& rational::operator~(void)
{
 int n1,n2;
 rational* r;

 if(numarator!=0) {
// daca numaratorul este nenul se va efectua operatia
     n1=numarator;
     n2=numitor;
 }
 else {
// in caz contrar numarul este lasat neschimbat
     n2=numarator;
     n1=numitor;
 }

 r=new rational(n2,n1);

 return *r;
}

void main(void)
{
 rational a(1,2),b(3,4);

// stergere ecran
 clrscr();

 a.tip("a");
```



```
a=~a;
a.tip("a");

a=~a;
a.tip("a");

b.tip("b");

b=~~~~~b;
b.tip("b");

b=~b;
b.tip("b");

a=a*b;
a.tip("a");

b=a*b;
b.tip("b");

 a=~a*~b; // ATENTIE: precedenta lui "~" este mai mare decit
          // cea a lui "*", deci "~" se vor executa inaintea lui "*"

  a.tip("a");
}
```

În urma execuţiei programului se vor afişa următoarele rezultate:

```
Nr. rational  a = 1/2
Nr. rational  a = 2/1
Nr. rational  a = 1/2
Nr. rational  b = 3/4
Nr. rational  b = 4/3
Nr. rational  b = 3/4
Nr. rational  a = 3/8
Nr. rational  b = 9/32
Nr. rational  a = 256/27
```

Ceea ce ar trebui totuşi evidenţiat este modul în care se efectuează operaţiile. Deoarece se doreşte o implementare cît mai reală, rezultatul nu va fi memorat în primul operand, ci va fi returnat ca entitate de sine stătătoare. Avînd în vedere această necesitate, rezultatul va fi păstrat într-o zonă de memorie alocată dinamic. Acest subterfugiu are două avantaje:

a. dacă rezultatul ar fi fost o variabilă locală metodei "operator *", valoarea

s-ar fi pierdut în urma părăsirii funcţiei în cauză;

b. dacă rezultatul ar fi fost păstrat într-o variabilă statică a metodei "operator*", am fi putut avea neplăceri în cazul unor înlănţuiri de operatori de genul "m=a*b*c*d";

După cum am mai spus, reluăm exemplul precedent, cu unica modificare că operatorii vor fi de astă dată funcţii "friend".

Exemplul 3

```
# include <iostream.h>
# include <conio.h>

class rational{
     int numitor,numarator;
 public:
     rational(int n1,int n2);

     void tip(char *);

     friend rational& operator*(rational&,
                                rational&);
     friend rational& operator~(rational&);
};

rational::rational(int n1,int n2)
{
 numarator=n1;
 numitor = n2!=0 ? n2 : 1;
}

void rational::tip(char * nume)
{
 int a;
 cout<<" Nr. rational   "<<nume<<" = "<<numarator
    <<"/"<<numitor<<"\n";
}

rational& operator*(rational& n,rational& m)
{
 rational * r;
 int n1,n2;

 n1=m.numarator*n.numarator;
 n2=m.numitor*n.numitor;
```



```
 r=new rational (n1,n2);

 return *r;
}

rational& operator~(rational& m)
{
 int n1,n2;
 rational* r;

 if(m.numarator!=0) {
    n1=m.numarator;
    n2=m.numitor;
 }
 else {
    n2=m.numarator;
    n1=m.numitor;
 }

 r=new rational(n2,n1);

 return *r;
}

void main(void)
{
 rational a(1,2),b(3,4);

 clrscr();

 a.tip("a");

 a=~a;
 a.tip("a");

 a=~a;
 a.tip("a");

 b.tip("b");

 b=~~~~~b;
 b.tip("b");
```

```
    b=~b;
    b.tip("b");

    a=a*b;
    a.tip("a");

    b=a*b;
    b.tip("b");

    a=~a*~b;        // ATENTIE: precedenta lui "~" este mai mare
                    // decit cea a lui "*"
    a.tip("a");
}
```

În secţiunile ce urmează vom prezenta, pe rînd, cîteva cazuri concrete de operatori a căror redefinire necesită o atenţie deosebită, datorată unui comportament mai puţin obişnuit. Cu excepţia operatorului "=" (vezi "5.6."), ABSOLUT TOŢI CEILALŢI se moştenesc conform regulilor obişnuite.

5.4. Operatorii "new" şi "delete"

Redefinirea acestor doi operatori implică mai multe restricţii:

a. "new" trebuie să returneze "void *" şi să aibă ca prim parametru un "size_t" (tip de dată definit în "stdlib.h" şi utilizat pentru manipularea lungimii zonelor de memorie destinate păstrării obiectelor);
b. "delete" trebuie să returneze "void", să aibă un prim parametru "void *" şi eventual un al doilea de tipul "size_t";
c. ambii operatori vor fi IMPLICIT şi OBLIGATORIU metode statice;
d. din precedenta condiţie reiese că "new" şi "delete" NU POT FI VIRTUALI.

Între atîtea restricţii, iată şi un fapt îmbucurător: operatorii standard "new" şi "delete" pot fi apelaţi atît în domeniul de vizibilitate al clasei ce-i redefineşte, cît şi în cel al altor clase derivate din aceasta. Acest lucru se poate realiza în două moduri: fie implicit (în caz că se crează/distrug obiecte de alt tip decît cel în cauză), fie explicit (prin formule de genul "::operator new" sau "::operator delete").

Vom da acum un exemplu foarte simplu, în care atît "new" cît şi "delete" vor fi redefiniţi.

Exemplu

```
# include <iostream.h>
```



```
class exemplu{
    int a,b,c;

// membrul "lungime" se va declara "static" pentru a putea fi accesat in
// cele doua metode statice "new" si "delete"
    static short lungime;

 public :
    exemplu(void)
    {
        cout<<"Apel constructor EXEMPLU\n";
        a=b=c=0;
    }

    ~exemplu(void)
    {
        cout<<"Apel destructor EXEMPLU\n\n";
    }

// se utilizeaza parametrul "lng" deoarece eventualele clase derivate din
// "exemplu" pot avea lungimi diferite de cea a clasei de baza
    void * operator new(size_t lng)
    {
       cout<<"Operatorul \"new\" redefinit !!!\n";
       lungime=lng;

// tipul argumentului (sir de caractere) este suficient pentru a specifica
// faptul ca se apeleaza operatorul standard "new"
        return (void *) new char[lng];
    }

    void operator delete(void * p)
    {
// se tipareste lungimea obiectului (in octeti)
        cout<<exemplu::lungime<<"\n\nOperatorul
           \"delete\" redefinit !!!\n";

// tipul argumentului (pointer la "char") este suficient pentru a specifica
// faptul ca se apeleaza operatorul standard "delete"
        delete[lungime] (char*) p;
    }
};
```

```
// variabilele membru statice trebuie initializate
short exemplu::lungime=0;

class derivat : public exemplu{
    double dbl;

 public :
    derivat(void)
    {
        cout<<"Apel constructor DERIVAT\n\n";
        dbl=12.34;
    }

    ~derivat(void)
    {
        cout<<"Apel destructor DERIVAT\n";
    }
};

void main (void)
{
 exemplu * ptr;
 derivat * ptd;

 ptr=new exemplu;
 delete ptr;

 ptd=new derivat;
 delete ptd;
}
```

Execuţia programului va determina tipărirea pe ecran a următoarelor date:

Operatorul "new" redefinit !!!
Apel constructor EXEMPLU
Apel destructor EXEMPLU

6

Operatorul "delete" redefinit !!!
Operatorul "new" redefinit !!!
Apel constructor EXEMPLU
Apel constructor DERIVAT

Apel destructor DERIVAT
Apel destructor EXEMPLU

14

Operatorul "delete" redefinit !!!

O observaţie importantă este legată de setarea valorii parametrului "lng": el nu va fi setat de către programator, ci va fi dedus DE CĂTRE COMPILATOR în momentul apelului lui "new". Valoarea sa va fi egală cu numărul de octeţi necesar pentru memorarea unui obiect de tipul dat. Prezenţa unui astfel de parametru este ABSOLUT NECESARĂ, deoarece metoda "new" va fi moştenită şi de clasa "derivat", ale cărei instanţieri au o altă lungime. Distrugerea obiectelor create dinamic se va face corect tot prin grija compilatorului.

5.5. Operatorul"()"

Redefinirea operatorului "apel de funcţie" ne aduce în situaţia de a crea un operator binar ne-static de forma

expresie(lista_de_expresii)

Utilizarea funcţiei-operator "()" are două mari avantaje:

a. EVALUAREA şi VERIFICAREA listei de argumente se face întocmai ca la o funcţie obişnuită.
b. Modul de funcţionare al mecanismului de apel. Deşi OPERATORUL este binar, FUNCŢIA-OPERATOR poate avea oriciţi parametrii (cel de al doilea operand fiind o listă de argumente). Deoarece lista de parametrii poate fi vidă, înseamnă că al doilea operand poate lipsi cu totul.

În exemplul de mai jos, am folosit operatorul "()" pentru implementarea unui iterator (funcţie de parcurgere a unei liste într-o ordine prestabilită).

Exemplu

```
# include <iostream.h>
# include <process.h>

struct elem
{
 int nr;
 struct elem * urm;

 elem(int n,struct elem * p) : nr(n) , urm(p) {}
```

```
};

// aceasta clasa este destinata pastrarii si manipularii de liste
// simplu inlantuite de elemente de tipul "elem"
class list{
 struct elem * cap,* curent;
// "cap" pastreaza un pointer la primul element al listei, iar
// "curent" pozitia la care s-a ajuns in cadrul operatiei de iterare

 public :
     list(void) { cap=NULL; }

     void init(int);

// prototipul celor doi iteratori
     struct elem * operator() (void);
     struct elem * operator() (int);
};

// creaza o lista cu "n" elemente
void list::init(int n)
{
 if(n<1)
 {
     cout<<"Initializare eronata !!!\n\n";
     exit(1);
 }

 for(int i=0;i<n;i++)
 {
     struct elem * p;
     p=new elem(i,cap);
     cap=p;
 }

// la inceput, pozitia curenta reprezinta primul element
 curent=cap;
}

// asigura o parcurgere ciclica a listei (in urma terminarii
// unei parcurgeri, se va incepe din nou cu primul element )
struct elem * list::operator() (void)
{
```



```
 struct elem * p=curent;

 curent = curent->urm!=NULL ? curent->urm : cap;
 return p;

}

// forteaza inceperea iterarii cu elementul avind numarul de
// ordine "pozitie"
struct elem * list::operator() (int pozitie)
{
 curent=cap;

// parcurge lista (eventual de mai multe ori, caci "pozitie" poate fi mai
// mare decit numarul de elemente din lista) si se pozitioneaza pe
// elementul dorit
 for(int i=0;i<pozitie-1;i++)
     (*this)();

 return (*this)();
}

void main(void)
{
 list l1;

 l1.init(4);

 for(int i=0;i<13;i++)
     cout<< l1()->nr << " ";

 cout<< "\n" << l1(12)->nr <<"\n";

 for(int j=0;j<13;j++)
     cout<< l1()->nr <<" ";

}
```

În urma execuţiei, pe ecran se vor afişa rezultatele:

```
3210321032103
0
3210321032103
```

5.6. Operatorul "="

Redefinirea operatorului ne-static "=" pune probleme cu totul deosebite. Acest fapt se datoreşte mai multor proprietăţi neîntîlnite la ceilalţi operatori.

Cu riscul de a ne repeta, vom începe prin a reaminti că tuturor claselor ce nu-l redefinesc pe "=" compilatorul le ataşează IMPLICIT o astfel de metodă. Ea va reprezenta un operator binar avînd ambii operatori de acelaşi tip. Acest operator atribuie tuturor variabilelor membru ale primului operand valorile membrilor corespunzători din cel de al doilea. Practic, va fi vorba de o copiere bit-cu-bit a tuturor variabilelor membru ale respectivei clase.

O a doua remarcă se referă la faptul că operatorul "=" NU ESTE NICIODATĂ MOŞTENIT DE CLASELE DERIVATE.

În fine, ori de cîte ori o clasă X are ca membru un pointer "p" pentru care se alocă dinamic memorie, trebuie avut grijă să nu se ajungă într-una din situaţiile de eroare de mai jos.

Exemplul 1

```
class ex
{
 int * pointer;

 public:
    ex(int i) { pointer=new int; (*pointer)=i; }
    ~ex() { delete pointer; }
};

void functie(void)
{
 ex e1(10),e2(20);

 ex e3=e2; // EROARE: constructorul va fi apelat doar pentru
           // "e2", iar destructorul si pentru "e2" si pentru "e3"

 e1=e2;  // EROARE: zona de memorie deja eliberata de doua
         // ori (de catre destructorii apelati pentru "e2" si "e3") va
         // mai fi eliberata inca o data de catre destructorul apelat
         // pentru "e1"
}
```

Acest exemplu evidenţiază subtila deosebire care există între operaţiile de "asignare" (se referă la un obiect CREAT DEJA) şi "iniţializare" (se referă la acţiunile ce au loc în cadrul constructorului).

Există două situaţii speciale, în care nu va fi utilizat operatorul "=", ci constructorul de copiere ("X::X(X&)"):

- transmiterea unor obiecte ca parametrii actuali ai unei funcţii;
- utilizarea obiectelor ca valori returnate de către o funcţie.

În cele ce urmează, vom relua exemplul din secţiunea precedentă, avînd grijă ca, de astă dată, să nu mai lăsăm la voia întîmplării asignarea şi iniţializarea.

Exemplul 2

```
# include <iostream.h>
# include <process.h>
# include <conio.h>

struct elem
{
 int nr;
 struct elem * urm;

 elem(int n,struct elem * p) : nr(n) , urm(p) {}
};

class list{
    struct elem * cap,* curent;

 public :
    list(void) { cap=NULL; }
    list(list&); // INITIALIZARE: constructor de copiere

    ~list(void);

    void init(int);

    void destruct(void);
    void construct(list&);

    struct elem * operator() (void);
    struct elem * operator() (int);

    void operator= (list& l) // ASIGNARE
    {
       if(this==&l) return; // ATENTIE la forme de
                            // genul "a=a"!
```

```
    // In situatii de genul "a = a", in urma distrugerii vechiului obiect,
    // reconstructia lui nu mai are sens !

        destruct();
        construct(l);
    }
};

list::list(list& l)
{
 construct(l);
}

void list::construct(list& l)
{
 struct elem * tmp=l.cap;

 cap=NULL;

// copiere de lista simplu inlantuita
 while(tmp!=NULL)
 {
    struct elem * p;

    p=new elem(tmp->nr,cap);
    cap=p;
    tmp=tmp->urm;
 }

// pozitionare pe inceputul listei
 curent=cap;
}

void list::destruct(void)
{
 struct elem * p=cap,* t;

// distrugere lista simplu inlantuita
 while(p!=NULL)
 {
    t=p;
    p=p->urm;
    delete t;
```



```
 }
}

list::~list(void)
{
 destruct();
}

// metoda "init" e identica cu cea din sectiunea precedenta
void list::init(int n)
{
 if(n<1)
 {
    cout<<"Initializare defectoasa !!!\n\n";
    exit(1);
 }

// crearc lista simplu inlantuita
 for(int i=0;i<n;i++)
 {
    struct elem * p;
    p=new elem(i,cap);
    cap=p;
 }

 curent=cap;
}

// ambii operatori sint identici cu cei din sectiunea precedenta
struct elem * list::operator() (void)
{
 struct elem * p=curent;

 curent = curent->urm!=NULL ? curent->urm : cap;
 return p;
}

struct elem * list::operator() (int pozitie)
{
 curent=cap;
 for(int i=0;i<pozitie-1;i++)
    (*this)();
```

```
 return (*this)();
}

void main(void)
{
 clrscr();

 list L1;
 L1.init(4);

 list L2,L3=L1;
 L2.init(6);

 L3=L2;
 L2=L1;

 cout<<"Lucram cu \"L2\" :\n";

 for(int i=0;i<5;i++)
    cout<< L2()->nr <<"\n";

 cout<<"Lucram cu \"L3\" :\n";

 cout<< L3(3)->nr <<"\n";

 for(int j=0;j<7;j++)
    cout<< L3()->nr <<"\n";
}
```

Rezultatele afişate de program vor fi cele de mai jos:

Lucram cu "L2" :
0
1
2
3
0
Lucram cu "L3" :
2
3
4
5
0
1

2
3

5.7. Operatorul "[]"

Termenul englezesc "subscript" desemnează operatorul "[]", care este un operator binar ne-static. Al doilea operand al său ("indexul") poate fi, practic, de orice tip. O linie de program de genul "a[b];" va fi interpretată ca "a.operator[](b);".

Vom da acum un exemplu interesant de utilizare a operatorului "[]". Ideea programului este foarte simplă. Se va crea un şir, ale cărui elemente conţin informaţii legate de cîte o persoană (în cazul nostru datele personale vor fi numele şi greutatea). Se doreşte identificarea persoanelor în cauză după trei criterii bine precizate (nume, greutate şi număr de ordine în cadrul şirului). În acest scop, au fost creaţi trei operatori de tipul "subscript", avînd indexul de tipul pointer la caracter, respectiv număr real sau întreg. Toţi aceşti operatori vor căuta în şir elementul descris de valoarea în cauză. Dacă este găsit, se va returna un pointer la el. În caz contrar se va returna valoarea NULL.

Exemplu

```
# include <iostream.h>
# include <string.h>

class exemplu;

class persoana{
     char   nume[20];
     double greutate;

     friend exemplu;

 public :

// initializeaza un obiect de tipul "persoana"
     void init(char * p,double g)
     {
         greutate=g;
         strncpy(nume,p,19);
         nume[19]='\0';
     }
```

```
// tipareste informatiile dintr-un obiect "persoana"
     void tip(void)
     {
     // se testeaza valoarea lui "this" deoarece "tip()" poate fi aplicat
     // asupra rezultatului unui operator "[]" (care poate fi NULL)
          if(this!=NULL)
            cout<<"\n\tNume : "<<nume<<
                  " Greutate : " <<greutate;
          else cout<<"\nElement inexistent !!!";
     }
};

class exemplu{
     persoana sir[3];

 public :
     exemplu(void);
     persoana * operator[] (char *);
     persoana * operator[] (double);

     persoana * operator[] (int i)
     {
    // testeaza apartenenta indexului la domeniul de valori corecte
          if (i<3) return & sir[i];
          else     return NULL;
     }
};

// initializeaza cele trei elemente ale lui "sir"
exemplu::exemplu(void)
{
 for(int i=0;i<3;i++)
 {
     char   n[20];
     double d;
     cout<<"\nTastati un nume si o greutate : ";
     cin>>n>>d;
     sir[i].init(n,d);
  }
}

// parcurge tot sirul cautind un element cu numele "string"
persoana * exemplu::operator[] (char * string)
```



```
{
 for(int i=0;i<3;i++)
     if(strcmp(sir[i].nume,string)==0)
        return &sir[i];

 return NULL;
}

// parcurge sirul cautind un element cu greutatea "greutate"
persoana * exemplu::operator[] (double greutate)
{
 for(int i=0;i<3;i++)
     if(sir[i].greutate==greutate)
        return &sir[i];

 return NULL;
}

void main(void)
{
 exemplu e;

 e["Marta"]->tip();
 e[70.5]->tip();
 e[0]->tip();

 e["barta"]->tip();
 e[-23.1]->tip();
 e[10]->tip();
}
```

În urma execuţiei programului, se vor afişa rezultatele de mai jos:

```
Tastati numele si greutatea : Fuicu 80.9

Tastati numele si greutatea : Marta 65

Tastati numele si greutatea : Popa 70.5
      Nume : Marta   Greutate : 65
      Nume : Popa   Greutate : 70.5
      Nume : Fuicu   Greutate : 80.9
Element inexistent !!!
Element inexistent !!!
Element inexistent !!!
```

Făcînd cîteva modificări, programul ar putea fi mai apropiat de o implementare reală. Deorece s-ar fi pierdut din claritea exemplului, respectivele modificări le vom propune doar ca temă. Astfel, ar fi util ca lungimea şirului de persoane să fie variabilă (setabilă printr-un parametru al constructorului). În plus, dacă operatorul "[]" nu găseşte numele sau greutatea în cauză, se va crea un nou element avînd caracteristica respectivă. Acest element va fi adăugat în lista de persoane. Recomandăm rezolvarea problemei de mai sus prin două metode:

a. utilizînd liste înlănţuite;

b. utilizînd şiruri create dinamic.

5.8. Operatorul "->"

O caracteristică deosebit de importantă a acestui operator ne-static constă în UNARITATEA lui. O expresie de genul "a->m" se va interpreta ca "(a.operator->())->m". Din acest motiv funcţia-operator va trebui să returneze fie UN POINTER la o instanţiere a unei clase, fie UN OBIECT de un tip pentru care este redefinit operatorul "->". Aceste observaţii sînt logice, si pot fi deduse din forma "(a.operator->())->m", după cum urmează: "(a.operator ->())" este un apel de funcţie, iar asupra valorii returnate de către această funcţie se va aplica "->m". Deci, acest ultim "->" este fie operatorul standard (caz în care funcţia-operator "->" trebuie să returneze un pointer), fie tot un operator redefinit (situaţie ce implică returnarea unui obiect).

În continuare, vom analiza un exemplu în care s-au implementat ambele tipuri de operatori.

Exemplu

```
include <iostream.h>

struct ex
{
 int membru;
};

class exemplul_1{
     struct ex * pointer;

public :
     void set(struct ex & p) { pointer=&p; }

     struct ex * operator -> (void)
     {
```



```
        return pointer;
     }
};

class exemplul_2{
 exemplul_1 * pointer;

 public :
     void set(exemplul_1 & p) { pointer=&p; }

     exemplul_1 operator -> (void)
     {
         return *pointer;
     }
};

void main (void)
{
 struct ex  s;
 exemplul_1 e1;
 exemplul_2 e2;

 e1.set(s);

 e1->membru=3; // ATENTIE: ESTE UN APEL AL FUNCTIEI
               // "exemplul_1::operator-> ()"

 cout<<e1->membru<<"\n";// se va tipari valoarea 3 (in urma
                        // unui alt apel al aceluiasi operator)

 e2.set(e1);
 e2->membru=5;//ATENTIE: SINT DOUA APELURI DE FUNC-
              // TII-OPERATOR ("exemplul_2::operator-> ()"
              // si "exemplul_1::operator-> ()" )

 cout<<e2->membru<<"\n";//se va tipari valoarea 5 (in urma inca
                        //unui apel al acelorasi doi operatori)
}
```

5.9. Conversii

Există unele situaţii în care "overloading"-ul nu poate rezolva toate problemele. Spre exemplu, dacă dorim o implementare serioasă a clasei numerelor raţionale, ne vom izbi de o serie întreagă de inconveniente. În primul rînd ar fi efectuarea întregului set de operaţii (+, -, *, /, etc) nu numai între numerele raţionale, ci şi cu valori de alt tip (de exemplu adunări, scăderi ş.a.m.d. între numere raţionale şi întregi). Datorită lipsei de comutativitate a operatorilor redefiniţi, fiecare funcţie-operator trebuie scrisă de două ori (pentru ca ambii operanzi să poată fi de ambele tipuri). Dar, de aici , rezultă o multitudine de funcţii-operator, care nu fac nimic altceva decît să diminueze sensibil eleganţa abordării problemei.

Neajunsul ar putea fi foarte uşor rezolvat dacă s-ar crea un operator care să realizeze o conversie implicită a operanzilor de tip întreg în "raţional". Evident că, de fapt, nu se pune problema particulară a conversiei din întregi în numere raţionale, ci una mult mai generală: conversia unui tip dat "T1" într-un altul "T2".

Această problemă se poate diviza în două subprobleme:

a. Conversia unui tip oarecare "T1" într-un altul "T2", DEFINIT DE CĂTRE UTILIZATOR . Această subproblemă se va rezolva printr-un constructor al clasei "T2", care va avea UN SINGUR ARGUMENT (de tipul "T1"). De notat că, în această situaţie, nu are nici o importanţă tipul lui "T1": el poate fi atît predefinit ("int", "double" etc), cît şi definit de către utilizator.
b. Conversia unui tip T1 definit de către utilizator într-un tip T2 predefinit. Această situaţie specială va fi tratată de către operatori de tipul "T1::operator T2(void);", care vor realiza conversia şi vor returna o valoare de tipul T2.

Dacă cei doi operatori de conversie sînt definiţi, ori de cîte ori în cadrul unei instrucţiuni se va impune una sau cealaltă dintre conversii, se va apela IMPLICIT funcţia în cauză. Dar atenţie: CONVERSIILE NU VOR FI EFECTUATE DECÎT DACĂ SÎNT UNICE!!! Şi încă o problemă: toate obiectele create prin conversii (indiferent ca sînt implicite sau explicite) vor fi distruse de îndată ce nu mai este nevoie de ele!

Să analizăm acum un scurt program care implementează ambele tipuri de conversii.

Exemplul 1

```
# include <iostream.h>

class rational{
    int numitor,numarator;

public :
    rational(int n1,int n2=1)
    {
// Acest constructor poate fi utilizat si la conversii implicite
// de genul "int" -> "rational".

        numarator=n1;
        n2!=0 ? numitor=n2 : numitor=1;
    }

// conversie"rational-real"
    operator double(void)
    {
        return numarator/(double)numitor;
    }

    void tip(char * nume)
    {
        cout<<nume<<" = "<<numarator<<"/"
            <<numitor<<"\n";
    }

    rational& operator * (rational& r)
    {
        return *(new rational(numarator*r.numarator
                              ,numitor*r.numitor));
    }
};

void main(void)
{
 rational a(1,2),b(2,3),c(3),d(7,9);
 double dbl;

 a.tip("a");
 b.tip("b");
 c.tip("c");
 d.tip("d");
 c=a*b;
 c.tip("c");
```

```
 d=b*c*3;    // conversie "intreg -> rational" : numarul intreg 3 se
             // va converti intr-unul rational prin apelul
             // constructorului "rational(3,1)";

 d.tip("d");

 dbl=d; // conversie "rational->real" : "d" se va converti intr-un real
 cout<<"dbl = "<<dbl;
}
```

Rezultatele programului vor fi afişate pe ecran, după cum urmează:

```
a = 1/2
b = 2/3
c = 3/1
d = 7/9
c = 2/6
d = 12/18
dbl = 0.66667
```

În acest moment ne putem permite o analiză mai serioasă a situaţiilor în care este de preferat utilizarea funcţiilor "friend" celor de tip membru. Vom începe prin a reaminti că, indiferent de context, constructorii, destructorii şi funcţiile virtuale TREBUIE să fie METODE.

Există însă o situaţie în care nu pot fi utilizate decît funcţiile "friend":

```
class exemplu{
 public:
     exemplu(char *);
     void METODA(void);
     friend void Functie_Friend(exemplu);
};

// ...

void main(void)
{
 char string[10];
// ...
 string.METODA(); // EROARE: nu se incearca o
                  // conversie implicita"char*" -> "exemplu"
 Functie_Friend(string); // CORECT: se incearca conversia
                         //   implicita "char*" -> "exemplu"
}
```


După cum reiese şi din comentarii, diferenţa între cele două apeluri constă în aceea că o METODĂ poate fi apelată doar de un OBIECT DEJA EXISTENT, pe cînd FUNCŢIILE "FRIEND" pot fi apelate şi de OBIECTE ÎNCĂ INEXISTENTE, dar CARE POT FI OBŢINUTE PRIN CONVERSII UTILIZATOR. În afara acestei situaţii deosebite (ţinînd de aplicarea conversiilor implicite) şi de cazul operatorilor redefiniţi care trebuie să fie comutativi, recomandarea de a folosi metode şi nu "friends" rămîne valabilă, chiar dacă uneori pare mai comodă utilizarea funcţiilor "friend".

Se impune o analiză a cazurilor în care folosirea conversiilor utilizator poate provoca apariţia unor ambiguităţi. Ele sînt sintetizate în exemplul de mai jos.

Exemplul 2

```
// STRUCTURI DE DATE NECESARE EXEMPLIFICARII
// POSIBILELOR ERORI
struct ex1{
 ex1(int i) { /* ... */ }
};

struct ex2{
 ex2(int i) { /* ... */ }
};

struct ex3{
 ex3(float d) { /* ... */ }
};

struct ex4{
 ex4(ex3 e) { /* ... */ }
};

struct ex5{
 ex5(ex4 e) { /* ... */ }
};

// FUNCTII NECESARE EXEMPLIFICARII POSIBILELOR
// ERORI
void f1(ex1 e)
{ /* ... */ }

void f1(ex2 e)
{ /* ... */ }
```

```
void f2(ex5 e)
{ /* ... */ }

void f3(ex4 e)
{ /* ... */ }

void f4(float d)
{ /* ... */ }

void f4(ex3 e)
{ /* ... */ }

void main(void)
{
 f1(1); // EROARE DE COMPILARE. EXPRIMARE AMBI-
        // GUA nu se stie daca e vorba de f1(ex1(1)) sau f1(ex2(1))

 f1(ex1(1));    // CORECT
 f1(ex2(1));    // CORECT

 f2(12.34); // EROARE DE COMPILARE: compilatorul nu e
            // asigure trei conversii succesive
            // ( "double"-> "ex3"-> "ex4"-> "ex5" )

 f3(12.34); // CORECT: compilatorul e capabil sa asigure doua
            // conversii succesive ( "double"-> "ex3"-> "ex4" )

 f4(12);    // CORECT: conversie standard "int"-> "float"
}
```

După cum se vede, ambiguităţile sînt semnalizate în faza de compilare. Ele apar în două situaţii:

a. În momentul în care conversia nu este unică. Există însă şi o excepţie: în caz că se poate efectua o conversie standard, toate conversiile definite de către utilizator vor fi ignorate. Din acest motiv nu se semnalizează eroare la apelul lui "f4(12)".
b. În cazul în care pentru a se ajunge la tipul de date cerut este necesară aplicarea IMPLICITĂ a mai mult de două conversii utilizator.

Şi o ultimă observaţie: deşi conversiile utilizator sînt posibile, nu trebuie uitat că folosirea lor duce la obţinerea unor programe ilizibile, şi, uneori, chiar neportabile.

5.10. Problemă propusă spre rezolvare

Ca exerciţiu, va sugerăm să realizaţi o implementare a unei clase "matrice", care să permită utilizarea matricilor de întregi de dimensiuni variabile (pentru analogie, vezi clasa "BASIC_ARRAY" din secţiunea "3.11."). Manipularea elementelor matricii se va face cu ajutorul operatorilor "[]". Un exemplu de utilizare a clasei "matrice" ar fi secvenţa de mai jos:

```
void functie(int NumarLinii,int NumarColoane)
{
 matrice m(NumarLinii,NumarColoane);//declarare matrice

 for(int i=0,j;i<m.get_NrLinii();i++)
 {
    for(j=0;j<m.get_NrColoane();j++)// bucla de afisare
                                    // a unei linii
    {
       m[i][j]=i*10+j;      // initializare element
       cout<<m[i][j]<<" "; // afisare element
    }

    cout<<"\n";    // trecere la rind nou
  }
}
```

În cazul în care nu reuşiţi să rezolvaţi problema, vă recomandăm să încercaţi o implementare a clasei "matrice" pornind de la următorul schelet de program:

```
class linie{
     int* pTablou; // pointer la un array de intregi

 public:

// metoda de mai jos returneaza o referinta la elementul din
// array avind numarul de ordine "index"
     int& operator[] (int index)
      { return pTablou[index]; }

// metoda "construieste_linie" creaza dinamic un array de
// "NrColoane" elemente
  void construieste_linie(int NrColoane)
 {
     pTablou=new int[NrColoane];
```

```
 }
};

class matrice{
     linie* pMatrice; // pointer la un array
     int    NrLinii,NrColoane;

 public :

// functia operator de mai jos returneaza o referinta la un array de
//  intregi care reprezinta linia "index" a matricii de intregi
     linie& operator[](int index)
     {
      return pMatrice[index];
     }

     int get_NrLinii(void)
     {
      return NrLinii;
     }

     int get_NrColoane(void)
     {
      return NrColoane;
     }

     matrice(int,int);
};

// constructorul de mai jos creaza dinamic o matrice de
// dimensiuni variabile
matrice::matrice(int nrLin,int nrCol)
{
 NrLinii=nrLin;
 NrColoane=nrCol;

 pMatrice=new linie[NrLinii];

 for(int i=0;i<NrLinii;i++)
    pMatrice[i].construieste_linie(nrCol);
}
```


6. STREAMURI

6.1. Introducere

În acest capitol ne vom abate oarecum de la spiritul cărţii, în sensul că vom aborda o temă ce ţine mai mult de bibliotecile C++, decît de limbajul propriu-zis. Procedăm astfel pentru că, de fapt, STREAM-urile reprezintă o bibliotecă de funcţii de intrare şi ieşire, organizată însă într-o ierarhie riguroasă de clase. În orice caz, o trecere a lor în revistă se impune cel puţin din două motive: pe de o parte este vorba de un set de facilităţi inexistente în C-ul clasic, iar pe de alta constituie un excelent exemplu de eficienţă şi eleganţă în crearea unei ierarhii de clase.

În cele ce urmează, nu vom prezenta prima versiune a bibliotecii de stream-uri C++, ci o versiune mai nouă: biblioteca IOSTREAM. Aceasta îi este superioară celei vechi prin capacitatea ei de a utiliza toate caracteristicile versiunii C++ 2.0 (între care şi moştenirea multiplă). Pe moment, spaţiul nu permite o analiză exhaustivă a tuturor problemelor legate de stream-uri. De aceea, vă recomandăm că, după aprofundarea chestiunilor prezentate în carte, să efectuaţi un studiu atent al tuturor header-elor produsului BORLAND ce conţin informaţii legate de stream-uri.

Care e scopul STREAM-urilor? Aşa cum am mai spus, este vorba de o modalitate de manipulare a funcţiilor de intrare şi ieşire. Cu alte cuvinte, biblioteca de stream-uri ne pune la dispoziţie un set de metode care permit înlocuirea clasicelor "scanf" şi "printf". De ce le-am prefera pe cele noi? Foarte simplu: pentru că sînt mai elegante, flexibile şi eficiente. Şi, pe deasupra, graţie mecanismului de redefinire a operatorilor ("overloading"), vom putea crea un set propriu de metode, care să MANIPULEZE şi să VERIFICE toate datele aferente unor operaţii de intrare şi ieşire la nivel de OBIECTE. Majoritatea metodelor pot fi folosite incluzînd în programe fişierul "iostream.h". Altele vor necesita includerea lui "strstream.h" sau "fstream.h" (ambele includ la rindul lor pe "iostream.h").

După atîtea preparative, să vedem în sfîrşit ce e un STREAM. Specialiştii firmei BORLAND îl definiesc astfel: un STREAM este un CONCEPT ABSTRACT, care înglobează ORICE FLUX DE DATE de la o SURSĂ la o DESTINAŢIE ([BOR2], pag. 166). Dacă vrem să privim problema mai puţin teoretic, vom înlocui termenul de "flux de date" cu mai concretul "operaţii de citire şi scriere", şi termenul de "concept abstract" cu expresia "totalitatea mijloacelor de realizare". Evident că, de la caz la caz, sursa şi destinaţia vor putea fi ecranul, tastatura, un fişier sau o zonă de memorie. Putem reformula deci definiţia după cum urmează: STREAM-urile reprezintă

totalitatea modalităţilor de realizare a unor operaţii de citire sau scriere.

Ierarhia de clase din biblioteca "iostream" are două clase de bază: "streambuf" şi "ios". Prima este destinată operaţiilor de intrare/ieşire utilizînd un format minimal de citire şi scriere. Cea de a doua clasă conţine variabile de stare, metode necesare interfaţării cu "streambuf" şi un set de facilităţi destinate tratării erorilor ce pot să apară în timpul operaţiilor cu streamuri. "Streambuf" va fi moştenită de clasele "strstreambuf" şi "filebuf", pe cînd "ios" va da naştere unei arborescenţe de moştenire ceva mai complicate:

```
class  istream : virtual public ios;
class  ostream : virtual public ios;
class iostream : public istream , public ostream;

class  istream_withassign : public istream;
class  ostream_withassign : public ostream;
class iostream_withassign : public iostream;
```

Prefixele "i" şi "o" se referă la INPUT şi OUTPUT, adică la tipul stream-ului (de intrare sau ieşire). Deci clasele "istream" şi "ostream" permit tratarea caracterelor extrase/depuse din, respectiv în, obiecte de tipul "streambuf". Mai ciudat este cazul lui "iostream", care, îmbinînd caracteristicile primelor două, permite operaţii bidirecţionale (în sensul că aceeaşi entitate poate fi folosită atît ca sursă, cît şi ca destinaţie). Diferenţa dintre fii (cei analizaţi pînă acum) şi nepoţii clasei "ios" (cei care au în nume "_withassign") constă în faptul că aceştia din urmă mai posedă un operator suplimentar de atribuire, pe care nu-l vom analiza aici.

Trebuie să subliniem că "cin" şi "cout" (vă reamintiţi instrucţiunile de genul "cout<<alfa;") nu sînt nimic altceva decît instanţieri ale claselor "istream_withassign", respectiv "ostream_withassign". NOTA BENE: aceste două obiecte reprezintă echivalentele lui "stdin" si "stdout".

Pentru înlocuirea lui "stderr", firma BORLAND defineşte două obiecte ("cerr" şi "clog"), ambele instanţieri ale clasei "ostream_withassign". Unica diferenţă dintre ele constă în modul în care se execută "golirea" lor (în engleză "flushing mode"): în cazul lui "cerr" după fiecare inserare, la "clog" doar la umplerea buffer-ului. Pentru a clarifica această problemă, vom da o explicaţie mai amanunţită. Presupunem că fiecare dintre cele două obiecte are cîte un buffer intern, care conţine caracterele ce se vor afişa. În cazul lui "cerr", operaţiunile decurg astfel: la fiecare tentativă de scriere textul va fi depus în buffer, şi, indiferent că-l umple sau nu, va fi afişat. În cazul lui "clog", afişarea nu se va efectua decît dacă bufferul este plin. Mesajele de o lungime mai mică decît cea a buffer-ului nu vor fi tipărite singure, ci doar împreună cu alte mesaje ce vor veni după ele, pînă la umplerea buffer-ului.

6.2. Operaţii de intrare şi ieşire

Avînd în vedere similarităţile existente între "cin" şi "cout", vom începe tratarea lor în paralel, subliniind diferenţele de cîte ori va fi nevoie.

6.2.1. Introducere

Stream-urile de intrare şi ieşire redefinesc operatorii de shiftare ">>" şi respectiv "<<". În noua accepţiune, ei vor purta numele de EXTRACŢIE, respectiv INSERŢIE. Ambii sînt operatori BINARI, avînd drept prim operand (în partea stîngă) un obiect de tipul "istream" (respectiv "ostream"), cel de al doilea putînd fi orice tip pentru care operatorul a fost definit. Cei doi operatori sînt PREDEFINIŢI pentru tipurile de bază ale limbajului ("int", "char", "float" etc.) şi pot fi REDEFINIŢI pentru orice tip de date creat de către utilizator. Dacă redefinim ">>" sau "<<" ca avînd cel de al doilea operand de tipul "T" (tip definit de utilizator), operatorul se va numi "T extractor", respectiv "T insertor".

Analizînd secvenţa de mai jos, vom vedea un exemplu foarte simplu de aplicaţie utilizînd stream-uri:

```
char caracter;
cin>>caracter;
cout<<caracter;
```

Ce face această secvenţă de instrucţiuni? Va citi un caracter "caracter" din stream-ul de intrare (implicit tastatura) şi-l va scrie apoi în cel de ieşire (implicit ecranul).

Deoarece ambii operatori posedă proprietatea de asociativitate stînga şi, în plus, returnează o referinţă la primul lor operand (deci un "istream&", respectiv "ostream&"), ei pot fi înlănţuiţi. Acest fapt va permite o utilizare mult mai comodă a lor. ATENŢIE: nu se acceptă decît înlănţuiri de operatori de acelaşi tip ("<<" cu "<<" şi ">>" cu ">>"), nu şi înlănţuiri mixte ("<<" cu ">>")

Exemplul 1

```
# include <iostream.h>

void main(void)
{
 int a,c;
 float b;

 cout<<"Tastati un intreg si apoi un real !\n";
```

```
    cin>>a>>b;
    c=a+1;

    cout<<"a+b="<<a+b<<" a&c="<< (a&c) <<"\n";
}
```

În urma execuţiei programului, pe ecran se vor afişa următoarele rezultate:

```
12345.6789
a+b=357.678894 a&c=12
```

Ce se întîmplă în acest exemplu: sînt citite două numere, unul întreg şi unul real, după care se vor tipări rezultatele a două operaţii aritmetice. Ultima instrucţiune a lui "main()" merita privită cu atenţie. Deoarece operatorul redefinit "<<" îşi păstrează precedenţa, trebuie avut grijă că, în cazul în care încercăm tipărirea unor expresii ce folosesc operatori cu o precedenţă mai mică decît a lui "<<", să utilizăm parantezele (în cazul adunării nu era nevoie, dar era obligatoriu la "a&c").

Se mai cuvin făcute două observaţii:

a. "cin>>a>>b;" este echivalent cu

```
(cin.operator>>(a) ).operator>>(b);
```

b. utilizarea operatorului de shiftare "<<" în cadrul unei operaţii efectuate cu "cout" se poate face incluzînd operaţia în cauză între paranteze

```
cout<<"a<<2="<< ( a<<2);
```

6.2.2. STREAM-uri şi tipuri de date predefinite

Deoarece în cadrul acestei subsecţiuni se va analiza în detaliu funcţionarea operatorilor "<<" şi ">>", discutarea lor se va face separat.

6.2.2.1. STREAM-uri de intrare

a. EXTRACTORI pentru tipuri de date numerice

În această categorie intră numerele întregi şi reale ("short", "int" şi "long" cu sau fără semn, respectiv "float" şi "double"). Deoarece ele vor fi tratate asemănător, le vom explica împreună. Înainte de a analiza modul de funcţionare al mecanismului de extragere, trebuie să dăm o definiţie. Toate acele caractere pentru care funcţia de bibliotecă "isspace()" returnează o valoare nenulă se vor numi CARACTERE SPAŢIU (sau DE TIP SPAŢIU). Din această categorie fac parte - între altele - blancul, '\t' şi '\n'.

Revenind, a extrage o valoare numerică înseamnă:

- Ignorarea tuturor caracterelor de tip spaţiu.
- Un test dacă primul caracter ne-spaţiu este cifră, semn sau, eventual, punct zecimal (la reali). În caz că această condiţie este îndeplinită, se va continua cu următorul pas. Altfel, se va semnaliza o eroare şi nu vor mai putea fi efectuate operaţii de intrare pînă ce nu se tratează respectiva eroare.
- Se extrag toate caracterele consecutive, pînă la întîlnirea unuia invalid pentru respectivul tip de dată.
- În cele din urmă, se va efectua conversia string-ului în cauză la tipul de date respectiv.

b. EXTRACTORI pentru caractere

În cazul tipului de date "char", o secvenţă de genul

```
char c;
cin>>c;
```

atribuie variabilei "c" primul caracter NE-SPAŢIU aflat în stream-ul de intrare.

Dacă se doreşte extragerea unui şir de caractere, se vor citi TOATE CARACTERELE NE-SPAŢIU întîlnite consecutiv în stream-ul de intrare, pînă se va ajunge la un caracter SPAŢIU. String-ului astfel obţinut i se va adăuga (la sfîrşit) un caracter NULL. Trebuie tratat cu atenţie cazul în care string-ul de la intrare depăşeşte lungimea şirului ţintă. Cu un minim de cunoştiinţe relativ la formatele de intrare, astfel de probleme se evită foarte uşor (vezi secţiunea "6.4.").

Înainte de a încheia această subsecţiune, mai trebuie făcută o subliniere. Să presupunem că, la o operaţie de citire, primul caracter ne-spaţiu din stream-ul de intrare va fi INVALID pentru tipul de dată citit. În acest caz, valoarea variabilei - ţintă va rămîne neschimbată! În plus, se va seta indicatorul "fail" (eşec) a lui "cin", fapt care va duce la imposibilitatea de a mai citi din acest stream pînă la resetarea indicatorului. Mai multe amănunte despre indicatorul "fail" vom vedea în secţiunea "6.3." (în care vom analiza amănunţit toate problemele ţinînd de tratarea erorilor apărute în cadrul operaţiilor cu stream-uri). Pînă atunci, un scurt exemplu reprezentativ pentru astfel de situaţii:

Exemplu

```
# include <iostream.h>
```

```
void main(void)
{
 int var1=9,var2=8;

 cout<<var1<<" "<<var2<<"\n";//se vor tipari cele doua valori

 cin>>var1;      // vom tasta litera "m", fapt ce va duce
                 // la setarea lui "fail"

 cin>>var2;      // deoarece este setat flagul "fail", nu se va mai
                 // putea efectua citirea lui "var2"

 cout<<var1<<" "<<var2<<"\n"; // se vor tipari aceleasi doua
                              // valori : 9 si 8
}
```

Pe ecran vor fi afişate informaţiile de mai jos.

```
98
m
98
```

6.2.2.2. STREAM-uri de ieşire

Tipurile de date predefinite pe care le acceptă implicit un stream de ieşire sînt: "char", "short", "int", "long" (cu sau fără semn), "float", "double", "long double", "char *" şi "void*". Tipurile numerice vor fi afişate folosind convenţiile implicite de la "printf()" (cu condiţia de a nu fi modificat formatul de afişare). Astfel, în secvenţa de mai jos, cele două modalităţi de tipărire vor avea acelaşi efect:

```
int a;
long b;
char c;
double d;

// ...

cout<<a<<" "<<b<<" "<<c<<" "<<d<<"\n";
printf("%d %ld %c %g\n",a,b,c,d);
```

Pointerul la caracter este utilizat la tipărirea string-urilor, iar "void *" la cea a variabilelor de tip "pointer" (afişarea se va face în hexazecimal).

```
char a[]="program";
```



```
void * p=a;

cout<<"String = "<<a<<"      "<<"pointer = "<<p;
```

Efectul execuţiei acestei scurte secvenţe va fi afişarea pe ecran a următorului mesaj:

String = program pointer = 87F0:0000

6.2.3. Alte metode de manipulare a STREAM-urilor

Clasele "istream" şi "ostream" mai dispun de cîte două metode (simetrice ca mod de utilizare), care fac acelaşi lucru ca şi operatorii "<<" şi ">>". Ele se numesc "get" şi "read", respectiv "put" şi "write", şi vor fi analizate în cele ce urmează.

Diferenţa majoră între aceste funcţii şi cei doi operatori prezentaţi anterior constă în faptul că setarea diverselor formate de lucru (citire/scriere cu format special) nu influenţează decît pe cei din urmă.

Fie următoarea secvenţă:

```
char a[10];

// ...

cout<<a;
cout.write(a,10);
```

Ţinînd cont de cele afirmate mai sus, s-ar putea ca cele două funcţii să aibă efecte diferite, în funcţie de formatul cu care va realiza scrierea operatorul "<<".

a. **get**

Clasa "istream" posedă nu mai puţin de 6 metode purtînd acest nume. În cele ce urmează, ne vom limita la a vă prezenta două dintre ele.

Analizînd prototipul de mai jos, vom deduce că această metodă citeşte cîte un caracter din stream-ul de intrare şi-l înscrie în bufferul indicat de primul parametru. Numărul maxim de caractere care se citesc este dat de valoarea celui de al doilea parametru decrementat cu 1. Ultimul parametru va specifica aşa-zisul caracter de sfîrşit (dacă în stream-ul de intrare se va întîlni acest caracter, citirea se va opri chiar dacă încă nu s-a citit numărul maxim de caractere). ATENŢIE: caracterul de sfîrşit nu va fi pus în bufferul destinaţie şi nici nu va fi eliminat din stream-ul de intrare.

```
istream& istream::get(char *,int,int='\n');
```

Metoda "get" are şi o variantă mai simplă, prin care se citeşte un singur caracter. ATENŢIE: spre deosebire de citirea caracterelor cu ">>", în secvenţa

```
char c;
cin.get(c);
```

variabila "c" va primi valoarea PRIMULUI CARACTER din stream-ul de intrare, CHIAR DACĂ ACESTA ESTE DE TIPUL SPAŢIU.

b. **put**

Metoda simetrică lui "get" are următorul prototip:

```
ostream& put(char);
```

Scopul ei este de a realiza tipărirea unui singur caracter. Astfel, o secvenţă de genul

```
char c1='i';
int  c2='j';
// ...
cout.put(c1);
cout.put(c2);
```

va avea ca efect afişarea celor două caractere ('i' şi 'j').

c. **read**

Această funcţie oferă posibilitatea de citire a unui şir de caractere. Prototipul ei este următorul:

```
istream& read(char*, int);
```

Cu alte cuvinte, metoda "read" va avea doi parametrii: un pointer la un şir de caractere (care va desemna bufferul destinaţie) şi un întreg care precizează numărul de caractere de citit.

d. **write**

Metoda "write" permite scrierea unui şir de caractere (de lungime cunoscută) într-un stream, şi are următorul prototip:

```
ostream& write(char*, int);
```

Analizînd secvenţa de mai jos, totul pare banal.

```
char a[]="string";
cout.write(a,6);
```

Dacă furnizăm corect cei doi parametrii (pointerul la string şi numărul de caractere de tipărit), mecanismul merge strună. Totuşi, se impune o

observaţie: spre deosebire de operatorul "<<", "write" va tipări TOATE cele "N" caractere, chiar dacă în cadrul stringului va întîlni un caracter NULL.

Deşi mai există şi alte metode ale clasei "istream", în încheierea subsecţiunii, vom mai prezenta doar funcţia membru "putback()". Ea are un singur parametru, care este de tipul "char". După cum indică şi numele, metoda "putback" va încerca să pună în stream-ul de intrare caracterul primit drept parametru. În caz de nereuşită, eroarea va fi semnalizată prin setarea stării stream-ului la valoarea "fail" (vezi secţiunea "6.3.").

Metoda "putback" este deosebit de utilă în cazul în care trebuie citite de la tastatură toate caracterele consecutive care îndeplinesc o anumită condiţie. În astfel de situaţii, oprirea se va face în urma extragerii unui caracter necorespunzător, care nu va fi prelucrat în faza respectivă. Dar asta înseamnă fie că respectivul caracter trebuie repus în stream-ul de intrare, fie că el se va pierde. Dacă, în viitor, va fi necesară reanalizarea lui, pentru a nu-l pierde, se va folosi funcţia membru "putback".

Exemplul de mai jos subliniază exact cele spuse înainte. Funcţiile "CONDITIE1" şi "CONDITIE2" reprezintă criteriul de oprire din căutare (testeaza apariţia caracterului terminal). Procedura "extrage_string" va prelua cel mult "max" caractere consecutive ce îndeplinesc o condiţie dată.

Exemplu

```
# include <iostream.h>

typedef int (*pFct)(char);

// testeaza aparitia unui "#"
int CONDITIE1(char c)
{
 return c!='#';
}

//testeaza aparitia unui "&"
int CONDITIE2(char c)
{
 return c!='&';
}

// extrage din "cin" toate caracterele ce respecta o conditie
void extrage_string(pFct CONDITIE,
                    char * string,int max)
{
 char c;
```

```
int contor=0;

cin>>c;          // citirea primului caracter ne-spatiu

if(CONDITIE(c))
do
    {
        string[contor++]=c;
        cin.get(c); // citire caracter (poate fi si spatiu)
    }
while( CONDITIE(c) && contor<max );

string[contor]='\0';

cin.putback(c);
}

void main()
{
 char a[100];
 extrage_string(CONDITIE1,a,99);
 cout<<a<<"\n";

 extrage_string(CONDITIE2,a,99);
 cout<<a<<"\n";
}
```

6.2.4. Extractori şi insertori definiţi de utilizator

În cele ce urmează, vom da un exemplu de redefinire a operatorilor "<<" şi ">>" astfel încît să permită citirea şi scrierea de obiecte definite de către utilizator.

```
# include <iostream.h>

# define nl      "\n"
# define MAXIM    20

class exemplu{
     char   nume[MAXIM];
     int    virsta;
     double greutate;
```



```
 public :
     friend istream& operator>>(istream&,exemplu&);
     friend ostream& operator<<(ostream&,exemplu&);
};

istream& operator>> (istream& i,exemplu& e)
{
 i.read(e.nume,MAXIM-1); // se evita citirea a mai mult de
                         // "MAXIM-1" caractere

 e.nume[MAXIM-1]='\0'; // sfirsitstring

 i>>e.virsta;
 i>>e.greutate;

 return i;
}

ostream& operator<< (ostream& o,exemplu& e)
{
 o<<"\n\n FISA PERSOANALA  :  \n";
 o<<"\tNume      : "<<e.nume<<nl;
 o<<"\tVirsta    : "<<e.virsta<<nl;
 o<<"\tGreutate : "<<e.greutate<<nl;

 return o;
}

void main()
{
 exemplu el;

 cin>>el;
 cout<<el;
}
```

Evident că, într-un exemplu mai complet, functia ">>" mai poate avea încă două secţiuni:

a. Una de verificare a validităţii datelor de intrare (în sensul că vîrsta şi greutatea trebuie să fie pozitive, etc.).
b. Una de eliminare a "zgomotelor". De exemplu, dacă dorim ca formatul de intrare să fie de tipul

```
Nume : Istrate    Virsta : 24   Greutate : 70.5
```

trebuie eliminate din stream-ul de intrare string-urile "Nume :", "Virsta :" şi "Greutate :".

6.3. Tratarea erorilor apărute în operaţii cu stream-uri

Analizînd declaraţia clasei "ios", vom observa că există un set de indicatori de stare care păstrează toate informaţiile aferente erorilor apărute în cursul operaţiilor cu stream-uri. Aceşti indicatori sînt memoraţi la nivel de bit în cadrul variabilei membru "state".

```
class ios
{
// ...
 protected :
     int state; // pastreaza la nivel de bit valorile indicatorilor
                // de stare

 public :
      enum io_state
      {
         goodbit  = 0x00,// ultima operatie s-a efectuat corect

         eofbit   = 0x01, // semnalizeaza lipsa de caractere
                          // disponibile pentru citire

         failbit  = 0x02,// semnaleaza esecul ultimei operatii
                         // de intrare/iesire; stream-ul nu mai
                         // poate fi folosit pina ce flag-ul nu
                         // va fi sters

         badbit   = 0x04, // semnaleaza faptul ca ultima opera-
                          // tie de intrare/iesire a fost invalida;
                          // in urma resetarii flag-ului S-AR
                          // PUTEA ca stream-ul sa mai poata
                          // fi folosit

         hardfail = 0x08 // semnaleaza o eroare ireparabila
      };
};
```


Vom studia acum metodele de care dispune clasa "ios" pentru a trata erorile ce pot să apară.

Începem prin a aminti o primă familie de trei funcţii care nu au nici un parametru şi returnează un "int":

"eof": returnează o valoare nenulă dacă indicatorul "eofbit" este setat, şi 0 în caz contrar.

"fail": returneaz o valoare nenulă în caz că într-o operaţie de intrare/ieşire s-a înregistrat un eşec (este setat unul dintre flagurile "failbit", "badbit" sau "hardbit") şi 0 în rest.

"bad": funcţionează ca şi precedenţa, atît că semnalizează doar apariţia erorilor (deci nu ia în considerare flagul "failbit").

Metoda "good" nu are parametrii şi returnează tot un întreg: nenul dacă nici un flag de eroare nu e setat şi 0 în caz contrar.

Avînd acelaşi prototip ca şi "good", "rdstate" va returna starea curentă a flagurilor sub forma unui număr întreg. Altfel spus, această metodă va returna valoarea membrului "state".

În fine, "void clear(int=0);" setează variabila de stare "state" la valoarea parametrului actual. Pentru a seta un singur flag (lăsîndu-le pe celelalte nemodificate) se poate recurge la apelul :

```
my_stream.clear( my_stream.rdstate() | val_flag );
```

unde "val_flag" va lua una din valorile definite de tipul enumerat "io_state".

Este necesară o precizare deosebit de importantă: funcţia "clear" este capabilă să şteargă orice flag MAI PUŢIN "hardfail"!!!

Clasa "ios" redefineşte şi doi operatori unari: "!" şi "void *". Primul se comportă întocmai ca şi metoda "int fail(void)", iar cel de al doilea exact invers (returnează 0 în caz că unul din biţii în cauza e setat şi diferit de 0 în caz contrar). Graţie acestor doi operatori e posibilă scrierea unor secvenţe de genul:

```
char c;

while(cin>>c) // aici se utilizeaza IMPLICIT conversia
              // lui "istream" la "void *"
{
 cout<<c;

 if(!cout)     // nu este operatorul standard "!", ci cel redefinit
    cerr<<"Eroare la tiparire !!!\n";
}
```

În încheierea secţiunii, vom aminti că pentru scoaterea unui stream din starea de eroare este necesară îndeplinirea a două condiţii:

a. să dispară cauza ce a provocat eroarea;

b. să fie şterse flagurile ce semnalizează eroarea în cauză.

6.4. Operaţii de intrare şi ieşire cu format

6.4.1. Indicatorii de stare ai formatului

Pentru a înţelege mai bine problemele legate de formatele de citire şi scriere, vom arunca o privire asupra modului în care este declarată clasa "ios".

```
class ios{
// ...
 protected :
    long x_flag; // pastreaza (la nivel de bit) indicatorii de
                 // format
    int x_width; // numarul de caractere utilizate pentru
                 // afisarea unei valori pe ecran

 public :
    enum {
       skipws    = 0x0001, // ignora spatiile de la intrare

       left      = 0x0002, // la operatiile de iesire se face
                           // aliniere stinga

       right     = 0x0004, // la operatiile de iesire se face
                           // aliniere dreapta

       internal  = 0x0008, // aliniere dupa semn sau
                           // specificator al bazei

       dec       = 0x0010, // conversie in baza 10
       oct       = 0x0020, // conversie in baza 8
       hex       = 0x0040, // conversie in baza 16

       showbase  = 0x0080, // se va tipari si baza
       showpoint = 0x0100, // se va tipari punctul zecimal

       uppercase = 0x0200, // pentru afisarea literelor din
```



```
                           // cadrul numerelor reprezentate in baza 16 se vor
                           // utiliza majuscule

       showpos   = 0x0400, // intregii pozitivi prefixati de
                           // "+"

       scientific= 0x0800, // reali scrisi sub forma
                           // 9.99999E4
       fixed     = 0x1000, // reali scrisi sub forma 99999.9

       unitbuf   = 0x2000, // streamurile se "golesc" (afi-
                           // seaza) dupa fiecare inserare
       stdio     = 0x4000, // "stdout" si "stderr" se vor goli
                           // dupa fiecare inserare
    }; // SFIRSIT"ENUM"

// ...
}; // SFIRSIT "IOS"
```

La operaţiile de ieşire, este deosebit de important numărul de caractere utilizate pentru afişarea unei valori pe ecran. În scopul păstrării acestei caracteristici există variabila membru "x_width", pe care am descris-o în cadrul declaraţiei clasei "ios". Implicit, valoarea lui "x_width" este 0, ceea ce semnifică utilizarea unui număr minim de caractere, care să permită totuşi afişarea valorii în cauză. Există însă situaţii speciale (tabelări, rapoarte etc.), în care este necesară o tipărire pe coloane aliniate estetic. În acest scop se va utiliza metoda "width" cu prototipul

```
int ios::width(int);
```

Scopul ei constă în setarea lui "x_width" la valoarea parametrului actual. Metoda va returna continutul lui "x_width" inainte de apelul ei.

Dacă valoarea setată de funcţia "width" este insuficientă pentru afişarea informaţiei, se va utiliza valoarea implicită 0. În acest fel, se evită pierderile de informaţie rezultate din trunchiere.

Exemplu

```
int wdth=cout.width(25); // salvare "x_width" vechi si
                         // setare a noii valori la 25
cout<<"MESAJ 1";
cout.width(wdth);          // restaurare "x_width"
cout<<"MESAJ 2";
```

Trebuie amintit faptul că există şi o metodă cu prototipul "int width(void);". Ea returnează valoarea curentă a lui "x_width", fără a o mai modifica.

În legătură cu membrul "x_width", se mai cuvin făcute două observaţii:

a. În cazul în care valoarea lui este mai mare decît dimensiunea minimă necesară, se va efectua o aliniere la stînga sau la dreapta. Amănunte despre aceasta vom vedea în exemplul ce urmează.
b. După fiecare operaţie de scriere, "x_width" va fi setat implicit la 0. În cazul înlănţuirii de operatori "<<", valoarea iniţială a lui "x_width" nu va fi folosită decît PENTRU PRIMUL OBIECT TIPĂRIT. Celelalte se vor afişa cu valoarea implicită 0.

Variabila membru "x_flag" va păstra (la nivel de bit) indicatori ce stabilesc formatul de citire/scriere. Pentru setarea lor se va utiliza un set de metode definite în acest scop. Valorile din tipul enumerat vor fi utilizate ca parametrii ai metodelor respective. Trebuie să amintim că prin termenul englezesc de "format state flags" sînt desemnaţi indicatorii utilizaţi în stabilirea formatului de intrare/ieşire.

Vom prezenta acum cîteva metode care au drept scop stabilirea formatului de tipărire.

Exemplu

```
# include <iostream.h>

void main(void)
{
 double a=987.654;

 cout.fill('_');   // setare caracter de umplere
 cout.width(10);   // setare "x_width"

// setare aliniere stinga
 cout.setf(ios::left,ios::adjustfield);

 cout<<a<<"\n";        // afisare numar real

 cout.fill('>');    // setare caracter de umplere
 cout.width(15);    // setare "x_width"

// setare aliniere dreapta
 cout.setf(ios::right,ios::adjustfield);
```



```
 cout<<a<<"\n";        // afisare numar real
}
```

În urma execuţiei acestui scurt program, vom observa că, pe ecran, au fost tipărite string-urile

```
987.654___
>>>>>>>987.654
```

Putem sintetiza în cîteva cuvinte efectul celor două noi metode:

"fill": Are un argument de tip "char", care va deveni noul caracter de umplere. În caz că numărul de caractere necesare pentru afişarea unei valori este mai mic decît "x_width", spaţiile rămase goale vor fi umplute cu caracterul în cauză.

"setf": Are drept scop setarea unor flaguri. Primul parametru specifică noua valoare a indicatorului, iar cel de al doilea desemnează flagul ce urmează să fie modificat. Pentru a înţelege mai bine cum funcţionează "setf", este necesară descrierea a încă trei variabile membru ale clasei "ios":

```
class ios{
// ...

   public :
      static const long basefield;// "oct",
                                  // "dec", "hex"

      static const long floatfield;// "scientific",
                                   // "fixed"

      static const long adjustfield;
                              // "internal", "left" si "right"
// ...
};
```

Cel de al doilea parametru al metodei "setf" poate fi una din cele trei variabile membru prezentate mai sus. În dreptul fiecăreia din ele am trecut ca şi comentariu valorile pe care le poate lua.

6.4.2. Manipulatori

În exemplele precedente s-a văzut că modificarea caracteristicilor formatului ("x_width", flag-uri, caractere de umplere etc.) are un dezavantaj: nu permite înlănţuirea operatorilor "<<" care utilizează formate diferite. Eliminarea inconvenientului se poate face cu ajutorul unor funcţii speciale,

asemănătoare operatorilor, numite MANIPULATORI.

Vom relua acum exemplul precedent, în care înlocuim metodele obişnuite cu manipulatorii echivalenţi:

Exemplul 1

```
# include <iostream.h>
# include <iomanip.h>

void main(void)
{
 double a=987.654;

 cout<<setw(10)<<setfill('_')
     <<setiosflags(ios::left)<<a<<"\n"<<setw(15)
     <<setfill('>')<<setiosflags(ios::right)
     <<a<<"\n";
}
```

Efectul acestui program va fi absolut identic cu cel al precedentului, afişîndu-se aceleaşi string-uri:

```
987.654___
>>>>>>>>987.654
```

În cadrul secvenţei de mai sus, se poate deduce uşor scopul fiecărui manipulator (prin analogie cu primul exemplu). În continuare, vom încerca o scurtă sistematizare a categoriilor de manipulatori din "iomanip.h". Pentru aceasta îi vom împărţi în 5 grupe:

a. **dec, hex, oct**

Toţi cei trei manipulatori din această categorie se apelează fără nici un parametru.

Pot fi folosiţi atît la stream-uri de intrare, cît şi de ieşire. Prezenţa lor provoacă setarea flagului de conversie în baza 10, respectiv 16 sau 8.

b. **ws**

Nu are nici un parametru. Este specific stream-urilor de intrare, utilizarea lui declanşînd procesul de eliminare a caracterelor spaţiu.

c. **endl, ends, flush**

Nici unul din cei trei manipulatori nu se apelează cu vreun parametru.

Scopul lor este următorul:

- **"flush"**: goleşte (afişează) stream-ul;
- **"ends"**: inserează în stream un terminator de tip NULL;
- **"endl"**: afişează un stream, după ce în prealabil s-a inserat în el un caracter "newline".

d. **setiosflags, resetiosflags**

Au un singur parametru, de tip "long". Setează, respectiv resetează, flagul indicat de parametrul actual.

e. **setfill, setw, setprecision**

Au un singur parametru, de tip "int". Setează (la valoarea parametrului actual) caracterul de umplere, respectiv "x_width" sau numărul de cifre al părţii zecimale a numerelor reale. Se folosesc atît la stream-uri de intrare, cît şi la cele de ieşire.

Odată încheiată prezentarea manipulatorilor predefiniţi, vom relua Exemplul 1 pentru a arăta modul de creare a unui manipulator definit de utilizator:

Exemplul 2

```
# include <iostream.h>
# include <iomanip.h>

# define nl "\n"

ostream&
 manipulator_demonstrativ(ostream& o_stream)
{
 return o_stream<<nl<<nl<<"O noua entitate : "
                <<nl<<nl;
}

void main(void)
{
 double a=987.654;

 cout<<manipulator_demonstrativ<<setw(10)
     <<setfill('_')<<setiosflags(ios::left)<<a
     <<manipulator_demonstrativ<setw(15)
     <<setfill('>')<<setiosflags(ios::right)<<a;
}
```

În urma execuţiei programului, pe ecran se vor afişa mesajele:

O noua entitate :

987.654___

O noua entitate:

```
>>>>>>>>987.654
```

Utilizarea maipulatorilor este posibilă datorită faptului că există un operator redefinit "<<" avînd următorul prototip:

```
ostream& operator<< (ostream&
                (* PointerLaFunctie)(ostream&));
```

La fiecare folosire a unui manipulator, adresa acestuia va fi transmisă operatorului "<<" prin intermediul parametrului "PointerLaFunctie". În acest mod operatorul "<<" va putea apela funcţia în cauză.

6.5. Fişiere şi stream-uri

În această secţiune vom analiza modul de utilizare a stream-urilor în efectuarea operaţiilor de citire/scriere din/în fişiere. În acest scop, biblioteca de streamuri conţine trei clase speciale: "ifstream", "ofstream" şi "fstream". Ele sînt derivate din "istream", respectiv "ostream" şi "iostream".

Exemplu

```
# include <iostream.h>
# include <iomanip.h>

# include <fstream.h> // HEADERUL NECESAR
                      // LUCRULUI CU FISIERE

void eroare(char * pt)
{
 cerr<<"\n\nEROARE la incercarea de a deschide
    fisierul "<<pt;
 exit(1);
}

void copy(ofstream& destinatie,ifstream& sursa)
{
 char c;

 while( destinatie && sursa.get(c) ) // se folosesc cei
                                     // doi operatori
                                     // redefiniti(vezi "6.3.")
    destinatie.put(c);
}
```



```
void main(void)
{
 char sursa1[14],sursa2[14],destinatie[14];

 ifstream s1,s2;
 ofstream dst;

 cout<<"Introduceti numele celor 3 fisiere : "
    <<"Destinatie Sursa1 Sursa2\n\n";

 cin.read(destinatie,13);
 cin.read(sursa1,13);
 cin.read(sursa2,13);

 destinatie[13]=sursa1[13]=sursa2[13]='\0';

 s1.open(sursa1,ios::nocreate);
 if(!s1) eroare(sursa1);// utilizeaza operatorul redefinit "!"

 s2.open(sursa2,ios::nocreate);
 if(!s2) eroare(sursa2); // utilizeaza operatorul redefinit "!"

 dst.open(destinatie);
 if(!dst) eroare(destinatie);
 //s-a utilizat operatorul redefinit "!"

 copy(dst,s1);
 copy(dst,s2);

 s1.close();
 s2.close();
 dst.close();
}
```

Programul de mai sus încearcă să concateneze două fişiere ("sursa1" şi "sursa2"), păstrînd rezultatul într-un al treilea fişier ("destinaţie"). Din cîte se vede, modul de lucru este deosebit de simplu: se crează trei stream-uri ("s1", "s2", "dst"), care vor fi ataşate fişierelor corespunzătoare (prin metoda "open"). Folosind metodele deja cunoscute "get" şi "put", în funcţia "copy" se va realiza copierea unui fişier sursă în cel destinaţie. Cu ajutorul metodei "close", legătura logică dintre stream-uri şi cele trei fişiere va fi desfăcută.

În acest moment ar fi utile cîteva sublinieri:

a. Odată cu realizarea/desfacere legăturii stream-fişier, metodele "open" şi

"close" efectuează şi deschiderea (implicit în mod text) şi închiderea fişierelor în cauză.

b. În caz de eroare la deschiderea unui fişier, se va seta indicatorul de eroare al stream-ului respectiv (vezi "6.3.")

c. Fişierul odată închis, stream-ului i se va putea ataşa un alt fişier de lucru. Astfel, în exemplul nostru, secvenţa

```
// ...
dst.close();
dst.open(alta_destinatie);
// ...
```

ar fi întru totul corectă.

d. Dacă e cazul, metoda "open()" poate fi foarte uşor evitată graţie unui mecanism ce permite specificarea numelui fişierului ca parametru al constructorului stream-ului. De altfel, exprimarea

```
ofstream dst(destinatie);
```

este echivalenta cu

```
ofstream dst;
dst.open(destinatie);
```

În continuare, vom analiza mai pe larg metoda "open". Declaraţia ei în cele două clase "ifstream" şi "ofstream" este cvasi-identică:

```
void open(char* ,int=ios::out,
          int=filebuf::openprot); // OFSTREAM
void open(char* ,int=ios::in ,
          int=filebuf::openprot); // IFSTREAM
```

Primul parametru reprezintă un pointer la numele fişierului, iar al doilea este o valoare ce specifică modul de deschidere al fişierului. Lista tuturor valorilor posibile pentru acest al doilea parametru e definită în clasa "ios" sub forma

```
class ios{
// ...
 public :
    enum open_mode
    {
        in          = 0x01 , // fisier de intrare

        out         = 0x02 , // fisier de iesire

        ate         = 0x04 , // dupa deschidere, pozitia curen-
                             // ta va fi chiar sfirsitul fisierului
```



```
        app         = 0x08 , // deschis pentru apendare
                             // (scriere la sfirsit de fisier)

        trunc       = 0x10 , // daca fisierul exista, ii sterge
                             // continutul (optiune implicita in
                             // caz ca e vorba de un fisier de
                             // iesire si nu apare una dintre op-
                             // tiunile "ios::app" sau "ios::ate")

        nocreate    = 0x20 , // daca fisierul nu exista, se sem-
                             // nalizeaza eroare

        noreplace =   0x40 , // deschiderea esueaza daca
                             // fisierul exista

        binary      = 0x80     // deschidere in mod binar
    };
// ...

};
```

Sînt permise înlănţuirile specificatorilor modului de deschidere prin "SAU LOGIC" la nivel de bit. O linie de program de genul

```
ifstream f1("nume",ios::in | ios::nocreate);
```

este absolut corectă, şi va avea drept efect încercarea de a deschide pentru citire un fişier numit "nume". Încercarea va da greş în caz că fişierul nu există.

În fine, clasa "fstream" permite crearea unor stream-uri ce suportă atît operaţii de intrare, cît şi de ieşire. Altfel spus, graţie acestei clase, vor fi posibile secvenţe de genul:

```
fstream my_stream("filename.ext",
                  ios::in | ios::out);
streampos pozitie_citire,pozitie_scriere;
char c='b';
// ...
pozitie_scriere=my_stream.tellp();
my_stream.seekp(7,ios::beg);
my_stream<<c;
my_stream.seekp(14,ios::cur);
my_stream<<c;
// ...
```

```
pozitie_citire=my_stream.tellg();
my_stream.seekg(-9,ios::end);
my_stream>>c;
```

Ceea ce sare imediat în ochi, este prezenţa a două familii de funcţii-membru:

a. **"tellp", "tellg"**:

Returnează poziţia curentă de scriere ("Put") sau citire ("Get") în/din stream.

b. **"seekp", "seekg"**:

Servesc la modificarea poziţiei curente de scriere şi citire. Ele au un prim parametru întreg; acesta specifică numărul de caractere cu care se va modifica poziţia relativ la al doilea parametru (indică reperul faţă de care se va efectua modificarea). Valorile pentru al doilea parametru sînt definite în clasa "ios" dupa cum urmează:

```
class ios
{
//...
    public :
        enum seek_dir
        {
          beg = 0 , // inceput de fisier
          cur = 1 , // pozitia curenta in fisier
          end = 2   // sfirsit de fisier
        };
// ...
};
```

6.6. Probleme propuse spre rezolvare

1. Implementaţi un manipulator care să aibă drept scop testarea şi corectarea eventualelor erori ce pot apărea în cadrul operaţiilor de intrare /ieşire.
2. Redefiniţi operatorii "<<" şi ">>" de aşa manieră încît să implementeze operaţii de scriere şi citire prin intermediul interfeţei seriale a unui PC. Practic, cu ajutorul celor doi operatori, se vor putea efectua transmiteri de date (recepţie şi emisie) între două calculatoare.

ANEXA A

Lumea minunată a O.O.P.

Una dintre tendinţele cele mai interesante ale momentului constă în încercarea de a reuni "sub" o interfaţă grafică comună puterea mai multor O.O.L. ("Object Oriented Languages"). Astfel, din SMALLTALK se vor putea apela biblioteci scrise în C++ sau PASCAL obiectual. În acest sens trebuie subliniată iniţiativa IBM (orice s-ar spune, importanţa acestei firme este deosebit de mare) de a utiliza atît limbajul SMALLTALK cît şi C++. Avantajul ar consta în faptul că primul este folosit în aplicaţii informatice avînd interfeţe utilizator prietenoase (destinate în general unor utilizatori neinformaticieni), pe cînd al doilea "coboară" la nivelul aplicaţiilor legate direct de sistemul de operare (drivere de echipamente periferice, etc.).

Şi cum tot am ajuns în finalul prezentării, să aruncăm o privire sintetică asupra principalelor compilatoare obiectuale existente în acest moment (ianuarie 1992) pe PC-uri:

a. **TURBO C++**

DESTINAT:

- în general amatorilor, dar şi profesioniştilor dornici să dezvolte soft sub DOS

CARACTERISTICI:

- mediu de programare ieftin şi robust

PREŢ:

- aprox. $300

b. **BORLAND C++ 2.0**

DESTINAT:

- profesioniştilor

CARACTERISTICI:

- mediu de programare + TURBO DEBUGGER + TURBO ASSEMBLER + TURBO PROFILER + utilitar de conversie TURBO C 2.0 ->TURBO C++
- sistem evoluat de gestiune a memoriei în DOS
- poate exploata toate facilităţile WINDOWS; generează programe e-

xecutabile şi biblioteci de legare dinamică DDL specifice WINDOWS; are integrate toate funcţiile SDK Microsoft; posedă a o bibliotecă de sute de obiecte (inclusiv ferestre, meniuri, icon-uri, ...)

PREŢ:

- aprox. $800

c. **ZORTECH C++**

DESTINAT:

- de la amatori la profesionişti (există trei variante Standard, Developer şi Scientific)

CARACTERISTICI:

- asemănătoare cu BORLAND C++, dar NU ESTE MEDIU DE PROGRAMARE!!!
- ca organizare a ierarhiei de clase se apropie de SMALLTALK;
- versiunea Developer aduce primul compilator generînd cod specific unui I386-DX

PREŢ:

- aprox. $300 , $500 , $650

d. **SMALLTALK/V**

DESTINAT:

- în general profesioniştilor

CARACTERISTICI:

- mediu de programare
- limbaj pur obiectual, nu "obiectualizat" (ca C++)
- bibliotecă de 300 de clase conţinînd 97% din codul folosit pentru propria lui scriere (lipsesc doar rutinele de compilare şi gestiune a memoriei)

PREŢ:

- începînd de la $650 (în funcţie de versiune)

e. **TURBO PASCAL FOR WINDOWS**

DESTINAT:

- profesioniştilor (este fratele mai mare al lui TURBO PASCAL 6.0, destinat aplicaţiilor DOS)

CARACTERISTICI:

- mediu de programare "sub" WINDOWS
- bibliotecă de obiecte (TURBO VISION) mult mai completă decît BORLAND C++
- posedă ObjectWindows (utilitar de creare de interfeţe utilizator)

PREŢ:

- aprox. $380 (TURBO PASCAL 6.0: $250)

BIBLIOGRAFIE

1) [BOR1] BORLAND INTERNATIONAL
TURBO C++: Getting Started,
1990

2) [BOR2] BORLAND INTERNATIONAL
TURBO C++: Programmer's Guide,
1990

3) [ECK] Eckel Bruce
Using C++
Osborne/Mcgraw-Hill,
1990

4) [STRO] Stroustrup Bjarne
The C++ Programming Language
Addison-Wesley Series in Computer Science
1986

5) [MDLR] Canarache Ana, Breban Vasile
Mic dicţionar al limbii române
Editura Ştiinţifică
1974

6) Colecţia revistei "INFOPC", anul 1991